Leveraging Service-Oriented Business Applications to A Rigorous Rule-Centric Dynamic Behavioural Architecture

PhD Thesis

ALI ALQAHTANI

Software Technology Research Laboratory
Faculty of Computing Sciences and Engineering

De Montfort University
May, 2010

Thesis Summary

Today's market competitiveness and globalisation are putting pressure on organisations to join their efforts, to focus more on cooperation and interaction and to add value to their businesses. That is, most information systems supporting these cross-organisations are characterised as service-oriented business applications, where all the emphasis is put on inter-service interactions rather than intra-service computations.

Unfortunately for the development of such inter-organisational service-oriented business systems, current service technology proposes only ad-hoc, manual and static standard web-service languages such as WSDL, BPEL and WS-CDL [3, 7].

The main objective of the work reported in this thesis is thus to leverage the development of service-oriented business applications towards more reliability and dynamic adaptability, placing emphasis on the use of business rules to govern activities, while composing services. The best available software-engineering techniques for adaptability, mainly aspect-oriented mechanisms, are also to be integrated with advanced formal techniques. More specifically, the proposed approach consists of the following incremental steps. First, it models any business activity behaviour governing any service-oriented business process as Event-Condition-Action (ECA) rules. Then such informal rules are made more interaction-centric, using adapted architectural connectors. Third, still at the conceptual-level, with the aim of adapting such ECA-driven connectors, this approach borrows aspect-oriented ideas and mechanisms, and proposes to intercept events, select the properties required for interacting entities, explicitly and separately execute such ECA-driven behavioural interactions and finally dynamically weave the results into the entities involved. To ensure compliance and to preserve the implementation of this architectural conceptualisation, the work adopts the Maude language as an executable operational formalisation. For that purpose, Maude is first endowed with the notions of components and interfaces. Further, the concept of ECA-driven behavioural interactions are specified and implemented as aspects. Finally, capitalising on Maude reflection, the thesis demonstrates how to weave such interaction executions into associated services.

To My Wife and our Son Abdullah

Acknowledgment

As this work would not have been possible without the support of *Allah*, who took me through this mission and help me to accomplish it and the support I received from many different people, let me take this chance to express my gratitude to them.

First of all, I would like to mention my doctoral advisor Prof. Dr. Hussein Zedan . Being a member of his research group "Software Engineering", I could always seek his advice and benefit from his comprehensive knowledge of the research area. Regularly pushing for further improvements of my drafts, he substantially contributed to the quality of this thesis. Thanks also go to Dr. Amelia Platt and Dr. Martin Ward.

I would like to thank all members of the Software Technology Research Laboratory, Faculty of Computing Sciences and Engineering, De Montfort University for the friendly and convenient working atmosphere I could experience there.

I owe my mother, brothers and sisters an ocean of gratefulness for their continuous moral support and prayers. Definitely, I could not have finished this work without their help. I would like to express my appreciation to many special people who contributed to my work with their continuous encouragement. Among them, I wish to thank Dr. Mohamed Taye, Dr. Ali Albayati, Nasir Alalwan, Ahmed Alzahrani, Ahmed Alghamdi, Khalid Aldrawiesh, Yazed Alsaawy and finally Dr. Ahmed Alghamdi, Ajab Khan and Dr. Osama Elhassan who deserves special thanks. Without question and beyond any doubt, their moral support and suggestions were of great value.

My work has been accompanied and financially supported by Kingdom of Saudi Arabia. I feel very grateful for this support and wish the graduate school a successful continuation of their program for the benefit of many future Ph.D. students.

Last but foremost; I would like to thank my wife for never giving up her patience and warm-hearted encouragement.

Thank you all very much!

Ali Alqahtani

Leicester, May, 2010

Publications

[1] A. Alqahtani and H. Zedan. "A Rule-centric Architectural Conceptualisation for Service-Oriented Applications and its Aspectual Extension". *In First International Conference on 'Networked Digital Technologies'*, July 2009, Ostrava, Czech (NDT2009).

[2] A. Alqahtani and H. Zedan. "Aspectual Interactions for Adaptive Behavioral Web-Services with Tailored Maude-based Certification". *International Conference on e-Business and Information System Security 23-24,* May 2009 , Wuhan , China (EBISS 2009).

[3] A. Alqahtani and H. Zedan. "Agile Service-Oriented Applications Rule-based Foundation with Run Adaptability ". *JDCTA: International Journal of Digital Content Technology and its Applications*, *Vol*. 4, No. 1, pp. 16 - 25, February 2010,

Contents

1. Introduction and Motivation 13
1.1 Research Scope and State of the Art 15
1.1.1 Software-Engineering for SOA: Advances and Challenges 15
1.1.2 Research Questions 18
1.2 Research Objectives 19
1.3 Contributions of Thesis 20
1.4 Thesis Outline 22
2. Background 24
2.1 Service Technology: Overview of Existing Approaches 24
2.1.1 SOA and WS: Overview of Main Components 24
2.1.2 Web Services Standards 26
2.1.3 Service Technology and Ongoing Foundation 28
2.2 Business Rules and Interactions 30
2.2.1 BRs in Information Systems 30
2.2.2 Business Rules in Web-Services 32
2.3 Introduction to Architectural Techniques 33
2.3.1 Architectural Concepts 33
2.3.2 Architectural Descriptions Languages 35
2.3.3 Architectural Techniques in Software Evolution 36
2.4 Aspect-Oriented Techniques 37
2.4.1 Architectural Techniques and AOP 38
2.4.2 AOP in Web-Services 42
2.5 Maude 43
2.6 Service Technology and Foundation: Comparative Study 44
2.6.1 Comparison With Arsanjani's Approach 45
2.6.2 Comparison With Solanki's Approach 47
2.7 Summary 48
3. A Rule-centric Architectural Conceptualisation and its Aspectual Extension 49
3.1 Milestones and Steps of the Approach 51
3.2 Behavioural Services and Interactions With Event-driven Business Rules 53
3.2.1 ECA-driven Interaction Pattern for Activity Behaviour 53
3.2.2 Simplified Banking Example of The ECA-driven Pattern 55
3.3 ECA-driven Architectural Interconnections 57
3.3.1 ECA-driven Pattern For Architectural Service Interactions 58
3.3.2 ECA Architectural Pattern Applied to Banking 59
3.4 Towards Aspectual ECA-driven Architectural Interconnections 61
3.5 Summary 63
4. Compliant Maude-based Foundation and Validation of the Conceptual Modelling 65
4.1 Motivation 65
4.2 Extending Maude to Service Components and Interfaces 69
4.2.1 Service Components in Maude 70
4.2.2 Interfaces in Maude 72
4.2.3 Application on The Banking Example 73
4.3 Leveraging Maude to ECA-driven Behavioural Service Interactions 74
4.4 Runtime Adaptability of Aspectual ECA-driven Service Interactions 77

4.4.1 Dynamic adaptability of Services with Aspectual Maude: Informal Presentation..... 78
4.4.2 Dynamic Adaptability of Services with Aspectual Maude: Detailed Steps Execution..... 79
4.5 Summary..... 82
5. Validating the Approach with an E-commerce Case Study..... 83
5.1 E-commerce Application: Informal Presentation..... 85
5.2 The Developed Tool at Glance..... 88
5.3 The Order Activity: The Approach at Work..... 89
5.3.1 The Order Activity (Phase-rule): The ECA-driven Governing Rule..... 89
5.3.2 The Order Activity (Phase-int): The ECA-driven Architectural Model..... 90
5.3.3 The Order Activity (Phase-Maude): The Aspectual Maude at Work..... 91
5.3.3.1 The Service Components Formalised and Validated in the Extended Maude..... 91
5.3.3.2 The Required Service Interfaces Formalised and Validated in the Extended Maude..... 92
5.3.3.3 The Order ECA-driven Rule Formalised and Validated in the Extended Maude..... 94
5.3.3.4 The Order ECA-driven Rule and Dynamic Weaving Using in the Extended Maude..... 95
5.4 The Confirmation Activity: The Approach at Work..... 97
5.4.1 The Confirmation Activity (Phases-rule +Int): The ECA-driven Rule and its Architectural Modelling..... 97
5.4.2 The Confirmation Activity (Phase-Maude): Aspectual Maude at Work..... 99
5.4.2.1 The Service Components Formalised and Validated in the Extended Maude..... 99
5.4.2.2 The Required Service Interfaces Formalised and Validated in the Extended Maude..... 99
5.4.2.3 The Confirmation ECA-driven Rule Formalised and validated in the Extended Maude..... 101
5.4.2.4 The Confirmation ECA-driven Rule Formally and Dynamically Woven Using Maude..... 102
5.5 The Cancelling Activity: The Approach At Work..... 103
5.5.1 The Cancelling Activity (Phases-rule +Int): The ECA-driven Rule and its Architectural Modelling..... 103
5.5.2 The Cancelling Activity (Phase-Maude): Aspectual Maude at Work..... 104
5.5.2.1 Service Interfaces: Formalisation and Validation in the Extended Maude..... 104
5.5.2.2 The ECA-driven Cancel Rule: Formalisation and Validation in Extended Maude..... 106
5.5.2.3 The Dynamic Weaving of the Cancellation Rule Using Extended Maude..... 107
5.6 The Shipment Activity: The Approach at Work..... 108
5.6.1 The Shipment Activity (Phases-rule +Int): The ECA-driven Rule and its Architectural Modelling..... 108
5.6.2 The Shipment Activity (Phase-Maude): Aspectual Maude at Work..... 109
5.6.2.1 Service Interfaces: Formalisation and Validation in the Extended Maude..... 109
5.6.2.2 The ECA-driven Shipment Rule Using the Extended Maude..... 110
5.6.2.3 Dynamic Weaving of The ECA-driven Shipment Rule Using Extended

Maude..... 111
5.7 The Payment Activity: The Approach at Work..... 112
5.7.1 The Payment Activity (Phases-rule +Int): The ECA-driven Architectural modelling..... 112
5.7.2 The Shipment Activity (Phase-Maude): Aspectual Maude at Work..... 113
5.7.2.1 Service Components and Interfaces: Formalisation and Validation in the Extended Maude..... 113
5.7.2.2 Formalisation of the ECA-driven Payment Rule Using Extended Maude..... 114
5.7.2.3 The Shipment ECA-driven Rule Dynamic Weaving Using the Extended Maude..... 115
5.8 The Change-Profile Activity: The Approach at Work..... 116
5.8.1 The Change-profile Activity (Phases-rule +Int): The ECA-driven Architectural Modelling..... 117
5.8.2 The Change-Profile Activity (Phase-Maude): Aspectual Maude at Work..... 118
5.8.2.1 Service Components and Interfaces: Formalisation and Validation in the Extended Maude..... 119
5.8.2.2 The Formalisation Of ECA-driven Change-profile Rules Using Extended Maude..... 120
5.8.2.3 Dynamic Weaving Of ECA-driven Change-profile Rules Using Extended Maude..... 121
5.9 Summary..... 122
6. Conclusions and Future Work..... 124
6.1 Thesis General Consideration..... 124
6.2 Research Contribution..... 126
6.2.1 ECA-driven Modelling of Behavioural Service Features..... 126
6.2.2 Runtime Adaptability of Composite Service Behaviour..... 127
6.2.3 Compliant Operational Specification and Validation..... 127
6.2.4 Approach Assessment with Case-Studies..... 128
6.3 Future Work..... 128
6.3.1 The Enhancement of the Validation with Formal Verification..... 129
6.3.2 The Compliant Deployment Using Tailored Service Technology..... 129
6.3.3 Coping with Advanced Issues: Security and Context-Awareness..... 130
6.3.4 Investment of other Application Domains..... 130
References..... 134
Bibliography..... 143
Appendix
A Generic Maude Configurations and the Banking Case Study..... 144
A.1 Component Generic Configuration..... 144
A.2 Interface Generic Configuration..... 145
A.3 Coordination Generic Configuration..... 146
A.4 The Aspectual Maude-based Implementation for Banking Case Study..... 147
A.4.1 The Customer and Account Component in the Banking Example..... 147
A.4.2 Maude-based ECA-driven Interaction for Withdrawal..... 147
A.4.3 Dynamic Weaving Using Maude Reflective Strategy..... 150
B The Aspectual Maude-based Implementation for E-commerce Case Study.. 151
B.1 All The Components Involved in the E-commerce System..... 151
B.1.1 The Order Activity..... 154
B.1.2 The Confirmation Activity..... 159
B.1.3 The Cancel Activity..... 164

B.1.4 The Shipment Activity..... 168
B.1.5 The Payment Activity..... 170
B.1.6 The Change-profile Activity..... 176
C Introduction to Rewriting Logic and Maude..... 180
C.1 Maude: Main Features..... 180
C.2 Functional Modules..... 182
C.3 System and Object-Oriented Modules..... 186
C.3.1 Maude System Modules..... 187
C.3.2 Object-Oriented Module..... 190
C.3.3 Transforming Object-Oriented Modules into System Modules..... 193
C.4 Reflection and Internal Strategies..... 195
C.4.1 Reflection in Maude..... 195
C.4.2 Internal Strategies..... 198
C.5 Overview of Maude's Workstation Environment..... 200

List of Figures

2.1 Simplified SOA architecture..... 26
2.2 Operational rules with ECA..... 32
2.3 Graphical Illustration of Architectural Ingredients..... 35
2.4 Three-layer architecture..... 37
2.5 Component model of a banking system..... 40
2.6 The banking example using aspect components..... 41
3.1 Stepwise abstract ECA-based aspectual architectural model..... 53
3.2 ECA rule-centric architectural interactions..... 61
3.3 ECA-connector for standard withdrawal..... 62
3.4 ECA-connector for VIP withdrawal..... 62
3.5 Stepwise abstract aspect-orientation of ECA-based architectural model..... 65
4.1 Stepwise abstract aspect orientation of Maude ECA-based architectural..... 68
4.2 Dynamic adaptability of services using Aspectual Maude..... 80
5.1 Recapitulation of the phases of the proposed approach..... 87
5.2 The business activities of e-commerce case study and their ordering..... 88
5.3 The tool main functionalities at glance..... 90
5.4 ECA-driven pattern of e-commerce for order interaction..... 92
5.5 ECA-driven pattern of e-commerce for confirmation interactions..... 100
5.6 ECA-driven pattern of e-commerce for cancellation interactions..... 105
5.7 ECA-driven pattern of e-commerce for Shipment interactions..... 110
5.8 ECA-driven pattern of e-commerce for payment interactions..... 114
5.9 ECA-driven pattern of e-commerce for change profile interactions..... 119
5.10 ECA-driven pattern of e-commerce for regular change profile interactions. 120

Appendix

A.1 The Maude-based generic patterns of components..... 143
A.2 The Maude-based generic patterns of interfaces..... 144
A.3 The Maude-based generic pattern for ECA-driven coordinations..... 145
A.4 The Maude-based component for customer..... 146
A.5 The Maude-based component for account..... 147
A.6 The Maude-based interface for customer..... 147
A.7 The Maude-based account interface for withdrawal..... 148
A.8 The Maude-based ECA-driven coordination for withdrawal..... 148
A.9 The reflective strategy implementing the aspectual dynamic adaptive coordination..... 149
B.1 The Maude-based customer component..... 150
B.2 The Maude-based product component..... 151
B.3 The Maude-based shopping card component..... 151
B.4 The Maude-based history component..... 152
B.5 The Maude-based bank-card component..... 152
B.6 The Maude-based product for order interface..... 153
B.7 The Maude-based shopping card for order interface..... 154
B.8 The Maude-based customer for order interface..... 155
B.9 The Maude-based order coordination..... 156
B.10 The reflective strategy for order activity..... 157
B.11 The Maude-based customer for confirmation interface..... 158
B.12 The Maude-based product for confirmation interface..... 159
B.13 The Maude-based shopping card for confirmation interface..... 159
B.14 The Maude-based history for confirmation interface..... 160

B.15 The Maude-based confirmation coordination..... 161
B.16 The reflective strategy for confirmation activity..... 162
B.17 The Maude-based customer for cancel interface..... 163
B.18 The Maude-based product for cancel interface..... 164
B.19 The Maude-based history for cancel interface..... 164
B.20 The Maude-based cancel coordination..... 165
B.21 The reflective strategy for cancel activity..... 166
B.22 The Maude-based customer for shipment interface..... 167
B.23 The Maude-based history for shipment interface..... 167
B.24 The Maude-based shipment coordination..... 168
B.25 The reflective strategy for Shipment activity..... 169
B.26 The Maude-based customer for payment interface..... 170
B.27 The Maude-based account for payment interface..... 171
B.28 The Maude-based bank-card for payment interface..... 172
B.29 The Maude-based payment coordination..... 173
B.30 The reflective strategy for payment activity..... 174
B.31 The Maude-based customer for change profile interface..... 175
B.32 The Maude-based history for change profile interface..... 176
B.33 The Maude-based change profile coordination..... 177
B.34 The reflective strategy for changing profile activity..... 178
C.1 Maude divisions..... 180
C.2 Tress reduction Process..... 184
C.3 Rewriting process..... 188
C.4 UML of current account..... 190
C.5 Concurrent rewriting of bank accounts..... 192
C.6 Strategies control the rules execution..... 198
C.7 General view of Maude Workstation..... 199
C.8 The result split panel of Maude Workstation..... 200
C.9 The show information window..... 201
C.10 Window for sending reduction or rewriting commands to Maude..... 201

Abbreviations

ADL	Architectural Description Language
AOP	Aspect-oriented programming
AOSD	Aspect-Oriented Software Development
AO4BPEL	Aspect- Oriented For Business Process Execution Language
BPEL	Business Process Execution Language
BPEL4WS	Business Process Execution Language for Web Services
CBSD	Component-Based Software Development
CCS	Calculus Communication Systems
CPNets	Coloured Petri Nets
CSP	Communication Sequential Processes
C2	Chiron-2
DAML-S	DARPA Agent Markup Language for Services
DARPA	Defence Advanced Research Projects Agency
ECA	Event-Conditions-Actions
ITL	Interval Temporal Logic
HTTP	Hypertext Transfer Protocol
MASC	Manageable and Adaptive Service Compositions
OWL-S	Ontology Web Service for Services
Prolog	Program Logic
RL	Rewriting Logic
RuleML	Rule Markup Language
SAM	Software Architecture Model
SCA	Service-Component Architecture
SMTP	Simple Mail Transfer Protocol
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service-Oriented Computing
SOMA	Service-Oriented Modelling and Architecture
UDDI	Universal Description Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language
WSDL	Web Services Description Language
ViDRE	Vienna Distributed Rules Engine
WFN	Well-Formed Nets
WS	Web-Services
WS-CDL	Web Services Choreography Description Language
WSFL	Web Service Flow Language
W3C	World Wide Web Consortium
XLANG	Web Services for Business Process Design
XPath	XML Path Language
XSD	XML Schema Definition

Chapter 1

Introduction and Motivation

The current advances and seamless merging of (wireless) communication and computation, as well as that of individuals and society as a whole, are strongly reliant on software technology. Moreover, today's economy is highly competitive, increasingly evolving and globalised. As a consequence, organisations are more likely to focus on *loosely-coupled*, networked, cross-organisational and cooperative applications instead of their traditional integrated perception of centralised control.

These new organisational realities have put software-intensive applications at the centre of interest. More specifically, information systems aiming at (semi-) automating such cross-organisational realities are required to be *distributed*, *interaction-centric* and *adaptive*. Besides these essential requirements on interactions and adaptability, since organisations are becoming fully reliant on their information systems, *correctness* and availability are also major preoccupations while developing such systems.

Service-Oriented Computing (SOC) [83] has now become the finest emerging technology to introduce a breakthrough towards making these new requirements of business and agile inter-organisational realities (semi-) automating. Indeed, because this technology is a new computing paradigm, it focuses on tackling first-class and major driving principles of distribution, interaction, loose-coupling and heterogeneity. Web Services (WS), as the main enabling technology of service-oriented architecture (SOA), are self-contained, platform-independent software entities. WS are thus characterised through explicit *interfaces*, adapted to be universally described, published, composed, discovered and deployed over the Web [3]. Since basic services are not enough for requesters, *composition* is among the most important features of this service technology. Indeed, the composition of services allows the building of large-scale evolvable business (process) applications which can at run time solve realistic complex requests (i.e. business-to-consumer (C2B), Payers -to- Business (P2B) and Business-to- Business (B2B)).

As a technology, WS can be manipulated (e.g. described, published, discovered and composed) using sufficient criterions, figure it out in terms of XML-based languages, such as Web Services Description Language (WSDL), Universal Description Discovery and

Integration (UDDI), Simple Object Access Protocol (SOAP) [104], Business Process Execution Language for Web Services (BPEL4WS) [7] and Web Services Choreography Description Language (WS-CDL) [55]. These standardised languages are increasingly precocity in maturity and large adoption. Thereby, most world-wide cross-organisations are embracing WS to automate their networked business information systems.

However, with all this wide respect to these standards for SOC of sophisticated cross-organisational business applications, still difficulties that have been experienced by these standards exist and have not been solved. The focus of this thesis is to emphasise the three following challenging issues depending on intensified investigations among these standards' failures.

First, as is well known, most potential service-oriented applications such as e-commerce, e-health and e-government are the *knowledge-intensive* core (e.g. business rules govern the business behaviour) [81]. In contrast, all the above standards are unfortunately static and knowledge-scarce. For instance, WSDL allows just basic data such as message parameters and simple variables that make controlling of the activity flow and composing a given service-oriented business process which is the same with BPEL. That is, these standards do not permit the expression of the full business logic. Consequently, any governing rules must be hard-coded within the hidden service components, discouraging any form of competitiveness and adaptability. Externalise the business logic, thus allowing services to be adaptive and evolving is one important goals of this work and it is main objective that reported here.

Second, because cross-organisations are facing strong competition to shift their business to a *dynamic* environment where *unanticipated* events are likely, current service composition standards (e.g. BPEL, WS-CDL) are still exactly manual and static. In other words, besides ensuring adaptability at the design-time, it is also important to cope with runtime adaptability in an effective manner.

Third, most potential service-oriented applications, including e-banking, e-health and e-government, are becoming applications with critical mission. That is to say, they must express a high level of *dependability*, specially in view of correctness (and security). Otherwise, customers and partners will look for other similar but more reliable services.

Formal Methods [32] are thus much needed with service-oriented applications, such as this paradigm is increasingly maturing.

After this introduction concerning the potential advantages of service technology for developing loosely coupled cross-organisational applications, as well as an overview of some severe limitations of the current state of this technology, the remaining sections of this chapter are organised as follows. Section 1.1 sets out the general scope of the research, which aims to contribute to overcoming some of these shortcomings, within this emerging service-oriented paradigm. In particular, it sheds some light on the main software-engineering advances and techniques towards tackling such challenges. There follows a detailed account of the main research question and its different ramifications as the main push forces in reshaping the investigations reported in this thesis. In the next part, according to the research questions and objectives, the main original contributions of this work will be enumerated. The chapter ends by outlining the content of the remaining chapters.

1.1 Research Scope and State of the Art

The research will be approached first by looking further and more deeply into the potential and shortcomings of SOA in order to formulate the main research question. The literature review provides a clear view of all facets of this promising web-services technology and its governing SOA. Furthermore, the findings of this stage confirmed the following important observations. First, as aforementioned, *adaptability* and *dependability* are among the important factors for any cross-organisational alliance to stay competitive. Indeed, as noted above, with market globalisation and harsh competition, as well as networking advances, flexibility and reliability are the driving forces towards beneficial added value and success. Second, it is clear that (knowledge-based) *business logic*, reflected mainly in terms of *business rules*, represents the real backbone of most potential service-oriented business applications (e.g. e-commerce, e-health and e-government). There is thus a priori a promising research direction in the question of how to leverage service-oriented applications towards dynamic adaptability and correctness, in a disciplined way, where knowledge-intensive business logic is the main driving force.

1.1.1 Software Engineering for SOA: Advances and Challenges

If service-oriented applications are to be used to overcome such challenges, it is important to recognise the potential of advanced *software-engineering* methods and techniques. After extended bibliographical research, it was found that the following software-engineering concepts and techniques are the best adapted to taming the complexity, evolution and correctness of such complex software-intensive applications, including those based on the service-oriented paradigm.

***Architectural techniques for interaction-centric modelling**:* Software architectures [24, 70 and 105] promote interconnections to *first-class* citizens. Indeed, they permit the extraction of any code, within software or service components, which is responsible for *interactions*, into explicit and externalised *connectors*. By representing connectors explicitly in configurations, it becomes possible for services to evolve by operating only on the interactions between components and not on the intra-component computational codes [64].

***Business rules for behavioural and adaptability issues**:* Business rules [50, 106] are coined as constraints and policies for doing business in general. They are in particular introduced in terms of the occurrence of events, the application of constraints and the performance of associated actions (i.e. Event-Conditions-Actions or the ECA paradigm) [52]. As they are intuitive and sensitive to changes, most organisations attempt to separate them from computation [11]. Coincidently, as will be demonstrated in this thesis, business rules mostly involve several service components and thus regulate their compositions and interactions, rather than acting on the intra-component computations.

***Aspect-oriented techniques for runtime adaptability and separation of concerns**:* The aspect-oriented programming (AOP) [57] proposes factoring out *cross-cutting* concerns (e.g. interactions, logging, security, etc.). AOP thus allows cross-cutting concerns to be extracted from different code units (e.g. components, modules or classes) and externalised into so-called *advices*. Such advices, as factorised encapsulated behavioural units, can then be inserted accordingly into specific positions in the units concerned. While the right positions in which these advices have to be *woven* are referred to as *join-points*, the different ways of combining such advices before superposing them on the respective units are designated *point-cuts*. Principally, since service interactions are cross-cutting concerns, governing their

behaviour with (cross-service) business rules implies representing them as aspect-oriented advices, a notion which will be developed in this thesis.

***Formalisms for correctness and validation**:* As is well-known, correctness can only be completely ensured by the use of formal techniques. Indeed, testing remains a partial and ad-hoc method which cannot cover all cases. In the last two decades, many formalisms have been introduced and applied to specific (structural or behavioural) features to achieve more correctness and verification. Examples are (1) algebraic specifications [37] and rewriting techniques [36]; (2) Petri nets [85] and graph-transformation [49] as well-suited to system behaviour (e.g. work-flow, system dynamic).

Such advances in software engineering are adequate only for developing centralised software-intensive applications in general, however. That is, the field is still far from having produced a satisfactory approach which benefits from all these software-engineering potentials, leading towards a stepwise and disciplined development of the envisioned adaptive and loosely-coupled service-oriented applications. In particular, the following issues can be cited as serious limitations which also serve as strong motivations for this work.

***Architectural techniques do not benefit from business rules**:* Although several architectural description languages have been proposed, they have mostly been found to lack the interconnections to the flow of actions involved (from component interfaces) and their governing business rules. The work reported in this thesis thus aims to represent the capturing of event-driven business rules as behavioural connectors, allowing the promotion of adaptability within the service-oriented paradigm.

***Aspect-oriented mechanisms are not fully exploited at the architectural level**:* Although aspect-oriented mechanisms have been exploited at the programming level, it was found that little had been achieved during the early architectural phases, despite the potential benefits of this work. Firstly, the handling of cross-cutting concerns as *behavioural interactions* allows the promotion of understandability, transparency and adaptability. Moreover, the formulation of reasoning about interactions as "advices" ensures the production of correct and evolving specifications, which can later be efficiently implemented using any adapted AOP language.

***Lack of formalisms compliant with the above aspirations**:* Besides the lack of intuitive architectural approaches that benefit from business rules and aspect-oriented mechanisms,

the work presented in this thesis also argues that a sound foundation is essential to advance the reliable development of dynamically adaptive interaction-centric service-oriented systems. Indeed, the few existing aspect-oriented architectural proposals [14, 45, 46, 102] lack formal and highly declarative semantics. This hinders, among others, any realistic validation by rapid prototyping and /or sound verification. Lastly, adaptability (as advices) is addressed only at the design time, so the desired runtime adaptability is not supported.

1.1.2 Research Questions

On the basis of the above arguments concerning the general scope, research direction and objectives of this project, the main research question can be formulated as follows:

"How can service-oriented applications be leveraged towards rigorous and stepwise conceptual modelling, where dynamic adaptability becomes intrinsic and driven by the business logic and its governing compositional business rules?"

The work reported in this thesis aims to address this driving research question effectively. To do so requires its refinement into more manageable and tractable sub-questions. More precisely, the detailed sub-questions that are investigated in this thesis may be summarised as follows:

What is the proper way to cope with the business logic and its knowledge-intensiveness in potential service-oriented applications, in terms of intuitive identification and business-level descriptions?

What is the proper way to bridge the gap between such informal and business-level descriptions and subsequent more disciplined yet understandable conceptualisations, where essential service concepts such as service interfaces, service behaviour, service composition and adaptability come into play?

What is the proper way to leverage this disciplined conceptualisation towards a more operational rigorous formalisation, compliant with the outcome of the above sub-questions and allowing formal validation and verification and dynamic adaptability?

What is the proper way to validate the whole process through clear methodology, supported by realistic data from non-trivial case studies concerning adaptive and knowledge-intensive service-oriented applications?

Appropriate concepts, mechanisms and techniques will be proposed to resolve each of these sub-questions. To that end, this work will take advantage of advances in business rules, architectural techniques and aspect orientation as well as rewriting logic-based foundations. All these ingredients, as described subsequently, will be put together into an integrated proposal that caters for the development of adaptive and knowledge-intensive service-oriented applications in a progressive and disciplined manner.

1.2 Research Objectives

The overall objective of the present research is to tackle the abovementioned state-of-art shortcomings, while developing complex adaptive service-oriented business applications (e.g. e-banking systems, e-commerce applications, e-health applications). More precisely, the project has the following objectives.

***The development of business-level primitives**:* The purpose of such envisioned primitives is to facilitate the extraction, identification, characterisation and intuitive description of business rules in service-driven applications. This business-level elicitation phase is inspired by similar approaches in the domain of information systems [53, 106]. There is a particular focus on event-driven rules, as service technology is based on events and the exchange of messages.

***Integration of business rules with architectural interactions**:* In order to build highly (design-time) adaptive systems, this study proposes to capture architectural connector behaviours using event-driven ECA business rules. That is, instead of just externalising how services should interact by exchanging messages and actions, conditions and their triggering events are further externalised as business logic.

***Leverage of rule-centric architectural interactions with aspect-oriented mechanisms**:* An objective closely related to the first is to promote behavioural adaptability on service interactions at the design-time. Capitalising on the potentials of aspect-oriented mechanisms, particularly point-cuts and join-points, this thesis proposes ways to leverage such design-time architectural adaptability towards full runtime adaptability.

***Compliant formalisation fitting the above features**:* Starting from an intuitive architectural conceptualisation, it is argued that only on a sound foundation are correct, valid and

verifiable composite services feasible. Towards that aim, this thesis proposes ways to govern the envisioned aspectual and rule-centric architectural conceptualisation with an adapted formal setting. More precisely, the proposal uses Rewriting Logic (RL) [71] and its inherent Maude language [27]. ‘’ *Maude is a high-performance reflective language and system which supports equational and rewriting logic specification, computation and programming for a wide range of applications*’’[27]. The reasons for choosing this framework include: (1) RL is a unified framework for true-concurrent systems, thus promoting the ubiquitous distribution of service-oriented applications; (2) Maude is highly efficient, allowing millions of rewritings per second; (3) with their intrinsic reflection capabilities [28], RL and Maude promote separation and explicit control of rule executions using strategies. Moreover, Maude is extended here to fit the conceptual model proposed in this work.

Validation of the approach that has been proposed with realistic case-studies*:* To validate the envisioned approach at all stages (conceptual, formal and practical-levels), two potential areas are targeted, namely, banking and e-commerce. The objective thus consists in undertaking two applications from these fields and assessing the proposed approach at all stages.

1.3 Thesis Contributions

The main contribution of this thesis is to propose a stepwise and integrated architecture-centric approach for developing adaptive and knowledge-intensive service-oriented applications. The milestones and main features of this approach can be introduced as follows:

Rule-centric architectural model for *design-time adaptability* of service interactions: This study proposes to bring event-driven business rules to the architectural level. The interaction between service activities is externalised and abstracted from the computation within the business service, thus making them adaptable by construction. In other words, service designers and developers are able freely to change the service interactions (i.e. the rules governing the interactions) without changing or even knowing the computation within the services.

Aspect-oriented leveraging of the model for *runtime adaptability*: To facilitate the *runtime adaptability* of service interactions, any rule-centric architectural interaction is

treated as an *aspect*. This demonstrates how design-time adaptability can be smoothly leveraged towards fully runtime adaptability. Capitalising on aspect-oriented mechanisms, the events triggering any service interaction are first intercepted, then the required properties of the services involved are extracted via interfaces. Next, the associated rule-centric composition of services is performed. Finally, there is a demonstration explane how the resulting actions can be merged on the running services.

Compliant RL-based operational formalisation: Following a review of existing formalisms, rewriting logic was chosen as an appropriate operational means to formalise and validate an aspectual rule-centric conceptualisation. Among the reasons for this choice are the following: First, RL is a unified framework for true-concurrent systems and thus promotes the ubiquitous distribution of modern software-intensive systems in general and service-oriented ones in particular [74]. Second, Maude is highly efficient, allowing millions of rewritings per second. Third, with its intrinsic reflection capabilities [28], Maude promotes separation and explicit control of rule executions using strategies. Lastly, for certification purposes, RL has been endowed in recent years with a built-in LTL-based model-checker in Maude [38].

This thesis first proposes a way to formalise the behaviour of service components and interfaces in terms of event-driven business rules, by means of an adapted extension of the Maude language. Next, it presents a proposal to specify and execute any rule-centric architectural service interactions, using a further adaptation of Maude. Finally, recapitulating on the reflection capabilities of Maude, it shows how to dynamically (un)weave any aspectual rule-based service interactions on running Maude services.

Compliant Maude-based supporting tool for *validation and certification*: To support the approach at the level of Maude-based implementation and validation, software is developed to implement all the above-described extensions to Maude to fit the rule-centric aspectual architectural service interactions.

Realistic case studies dealing with *e-commerce*: To demonstrate the practicability of the proposed approach at a realistic level, two main case studies have been conducted. The first takes a simplified banking example as proof of concepts, then it is demonstrated how this approach can be scaled up to realistic applications. For that purpose, a typical service-

oriented e-commerce case-study was carried out by externalising its business logic at the composition level.

1.4 Thesis Outline

In accordance with the objectives of the present work, this section provides an overview of the remaining chapters of this thesis and summarises their content.

Chapter 2: Background

This chapter starts by reviewing the main ingredients of service technology and ongoing proposals for its foundation. It then introduces the three software-engineering topics on which the approach taken in this project is based. That is, as its purpose is to promote behavioural adaptively by extracting the business logic from service-oriented applications, the main definitions and concepts concerning business rules are first summarised. Emphasis will be given to their roles in naturally enhancing changes and adaptability in service-oriented applications. There is then an overview of architectural techniques, in particular of connectors, as first-class entities, and their role in taming design-time evolution in service-oriented applications. Since this thesis concentrates on runtime adaptability and separation of concerns, the main concepts and mechanisms of the aspect-oriented programming are summarised. Since the formal basis of the approach taken in this work is rewriting logic with its true-concurrent operational semantics and its implementation in the Maude language, an introduction to this language will be provide with an overview of the reflection mechanisms addressing runtime adaptability.

Chapter 3: A Rule-centric Architectural Conceptualisation and its Aspectual Extension

The chapter has two main objectives. First, it presents a conceptual model of rule-centric architectural interactions to address adaptive, knowledge-intensive, service-oriented applications in a disciplined manner. The model leverages event-driven (cross-entity) business rules to the architectural interconnections. That is, the behaviour involved in any business activity underlying a service-oriented business process is first described informally through associated business rules, then in a structured way using the ECA pattern. The rules are then upgraded as ECA-compliant rule-centric architectural connectors. The next step is to show how to provide this rule-centric architectural conceptualisation with aspect-oriented

mechanisms. The aim of this leveraging is to prepare this model to support not only design-time adaptability but also the non-intrusive runtime adaptability of the rule-centric interactions.

Chapter 4: Compliant Maude-based Foundation and Validation of the Conceptual Modelling

The first objective of this chapter is to explain how the Maude configuration should be adapted and extended so that services as well as their interfaces can be directly formalised, implemented and executed. The second is to show how the conceptual rule-centric interactions can be formalised in the extended Maude, then implemented and executed independently (using only required interfaces). The third is to propose a stepwise formalisation and implementation of the aspectual interactions, defined at the conceptual level, by recapitulating on the reflection capabilities of Maude. These Maude-based aspectual interactions permit a non-intrusive runtime interception of events, the execution of the respective interactions and the dynamic weaving of resulting behaviour into corresponding components.

Chapter 5: Validating the Approach with an E-commerce Case Study

In order to demonstrate the strong practicability of the approach, a case study is presented in Chapter 6. It encapsulates all activities in a typical e-shopping application, such as request, offer, confirmation, payment, cancelling and shipping. The chapter begins by informally describing all of the business activities underlying the case study, and then sets out the rules governing each activity. These are next expressed as architectural interactions and formalised in the extended Maude. Finally, their implementation and execution are described.

Chapter 6: Conclusions and Future Work

The concluding chapter in the beginning summarises the main contributions to the development of reliable runtime adaptive and knowledge-intensive service-oriented applications. Suggestions are then made for further improvements, extensions and explorations of the approach.

Chapter 2

Background

The reason of this chapter is to introduce the main service technologies and related software engineering concepts on which the present study is based. First, it lists the main components of service-oriented technology and web services, then reports on some ongoing proposals for the foundation of this service technology. Next, there is an overview of the main concepts underlying the business rules and their role in making service-oriented applications adaptive. The following section summarises techniques underlying architectural descriptions and their role in externalising interactions and business logic between any service entities as first-class primitives. Finally, since the project adopts aspect-oriented mechanisms for runtime adaptability, at the end of this the chapter a survey of the main concepts in this field will be introduced.

2.1 Service Technology: Overview of Existing Approaches

The objectives of this section are twofold. First, the main elements underlying service-oriented architecture (SOA) and its practical application to web service (WS) technology are reviewed. Second, since this thesis presents a formal conceptualisation, this section sketches the main ongoing formal approaches to service technology and WS.

2.1.1 SOA and WS: Overview of Main Components

Web services are characterised as *network-addressable* software units (e.g. components, modules, programs); that is, they are mostly developed to be used on the *internet*. Moreover, and in contrast to other internet-based applications (e.g. websites), WSs are exclusively accessible using well-defined and explicit *interfaces*. Such interfaces are further to be easily and *universally* exposed, invoked and *composed* to reflect to any complex service-oriented (business process) application as composed of several basic services. Nowadays, WS is widely known by its rich package of interoperable technologies and *standards* (e.g. SOAP, UDDI, BPEL, and UDDI) [83].

Cheng et al. [24] define WS as application-oriented software using specific web standards while serving *application-to-application* business processes. Sun [18], on the other hand, defines WSs as "*services offered through the web, where typically any business application sends a request to a service at a given URL using the SOAP protocol over HTTP. The service receives the request, processes it and returns a response*" [18]. IBM defines WSs as "*an application integration technology that can be successfully used over the Internet*" [103]. Another commonly referenced definition by the World Wide Web Consortium [3] sees a web service as *'' a software system identified by a URI which is designed to support interoperable machine-to-machine interaction over a network''* [3]. It has an interface that is capable of being described in a machine-processable format (specifically WSDL).

It is necessary to mention that WS have mainly emerged as platform-independent and composition-driven, intended to boost decentralisation and loosely-coupled cross-organisations [20, 81]. The service-oriented paradigm for software development over the internet is generally governed by the so-called "triangular" SOA. With respect to this generic architecture, web services represent the more widely adopted and practical instantiation of SOA, but are not the only way of using service technology. As depicted in the simplified Figure 2.1, SOA is based on three main principles: Publish-Find-Bind. That is, following the SOA architecture, to be used a service has to be published (not obligatorily over the internet), where subscribers can invoke it and finally bind it to others to build complex and composite services.

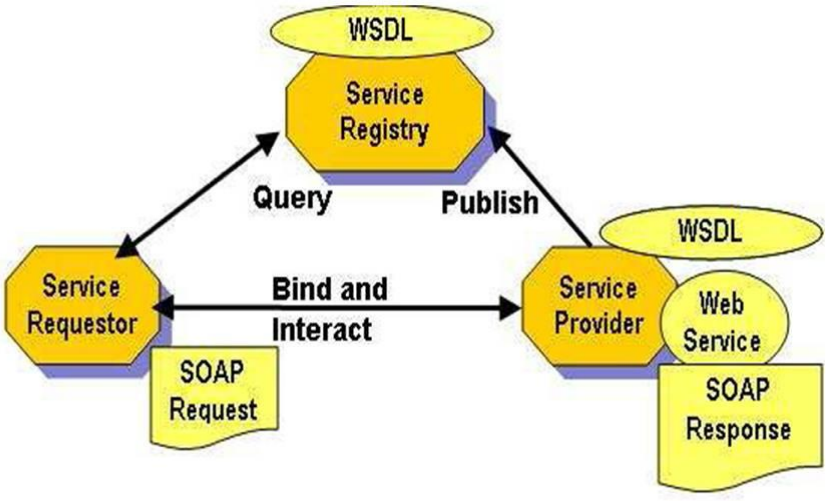


Figure 2.1: Simplified SOA architecture, [20]

SOA is one of an architectural style which fits internet-based business applications, as coordinated services which can be described, located and invoked over the network. As

depicted in figure 2.1, SOA is consist of three fundamental roles and three fundamental activities, the roles are: service provider, service requester and service broker; and three activities are : publish, discover, bind and invoke [59].

The service provider which is own the service from the business viewpoint, whereas from the architectural perspective, it defines the service details assumed to be published in one or more repositories (service registries) using the UDDI to be located / invoked by potential users.

The service requester is the business which requires a specific function to meet its needs. From an architectural view, the function of this application is to discover, bind and creates an interaction with a clear specific service. Different categories may contain the same service requesters, such as end users asking services through browser-based interfaces, application programs or other web services.

The service broker introduced a repository can be searched and contain any service description, which published their service descriptions the service providers. A service requester finds services and requests access to them by binding to the service provider.

2.1.2. Web Service Standards

The features of WS which prompted their adoption by most (cross-)organisational business applications relate to their standardised languages. Characteristics common to these standards include the following: (1) they all adopt XML as universal language; (2) they are all closely linked with the Web; (3) they all promote the exchange of messages in an event-driven manner; (4) and they all support heterogeneous protocols for communicating with each other without being dependent upon the implementation of the underlying platform. These standards are developed by many leading IT organisations including IBM, Microsoft and ARIBA, and are submitted to the World Wide Web Consortium (W3C). The most important WS standards include the following.

SOAP (Simple Object Access Protocol) is an XML-based message protocol which has a W3C specification. It provides the communication framework to transport XML-based messages anywhere across the net, in particular between web services and their clients [99]. SOAP is simple, extensible and platform-independent and is designed to send XML-based messages over the internet.

WSDL (Web Service Description Language WSDL), which is the next stage beyond SOAP-based messaging, is a specification that gives description to web services which are available to requesters. It is XML-based and describes the public interface of any WS. Service providers use WSDL to make advertisment and then declare or publish their services in the registry. The main constituents of a WSDL specification are the following.

Types: using XML-based type system such as XSD to define the data type that is contained.

Message: an abstract giving a type definition for the communicated data.

Operation: an abstract describing the service that supports an action.

Port Type: an abstract of any number of endpoints supporting a set of operations.

Binding: a particular port type with its concrete protocol and data format specification.

Port: a single endpoint expressed as a combination of a binding and a network address.

UDDI (Universal Description, Discovery and Integration) is an indexing web services business registry standard; thus development tools and applications can locate WSDL descriptions [80]. The communicates of UDDI is via SOAP and its role is like a directory to save WS information. The four types of information of UDDI XML schema which are defined are needed in order to use a partner's web service, these four typesare: *business entities*, *business services*, *binding templates* and *models*. Business entities show a business information including its name, description, services offered and contact information. Business services give deep details for each service that is being offered. Each service can have multiple binding templates, each one illustrating the service with its technical entry point (such as HTTP, SMTP). Finally, the model shows the specific protocol or standards a service uses.

BPEL (Business-Process Execution Language) builds on all the above basic standards and aims at *composing* elementary services into complex and thus realistic ones. Indeed, it is dificult to find a given basic service which will satisfy any meaningful customer request, so that composition is essential in service orientation. BPEL [4] is like workflows and business processes. It permits the partial ordering of different business activities to build service-oriented business processes (e.g. using sequence, choice, parallel, switch and other operators). The specification of (WS-)BPEL builds on IBM's Web Service Flow Language

(WSFL) [19, 61] and Microsoft's web service for Business Process design (XLANG) [97]. So, WS-BPEL assists peer-to-peer message exchanges sequencing, both asynchronous and synchronous, within stateful interactions involving two or more parties. It is used to model the behaviour of both executable and abstract processes. In addition to the so-called basic activities (send, receive and invoke), BPEL-like specifications include the following complex activities: (1) process activities sequencing, especially WS interactions, (2) engage of messages and process instances and (3) backup behaviour in cases of failure and exceptional conditions.

WS-BPEL located in the heart of WS architecture because it depends on the following XML-based specifications: XML Schema, WSDL and XPath. In this way, a WS-BPEL process definition introduce and/or uses one or more WSDL services and provides a description of the behaviour in shape of partially ordered activities. A WS-BPEL process definition express data variables, partners and process flow constructs. A partner join type characterises the conversational relationship between two services by give a defintion to the roles that played by each one of services in the conversation and determining the port types introduced by each role. It is important to mention that with BPEL, which is called orchestration-based composition (focussing on one service and calling the others), there is also the more global composition type called choreography, which can be specified by the standard WS-CDL [13, 30].

2.1.3. Service Technology and Ongoing Foundation

Due to unsure any level of correctness, reliability and flexibility of services as web standards, a number of foundations and methodologies have been proposed to overcome these serious shortcomings. Given the large of number of these ongoing formalisations and differences in motivation and focus, any exhaustive survey, comparison or fair assessment is outside the scope of this thesis, which will, however shed some light on selected service-oriented formalisms. These are explored during the first phase of this thesis.

To improve understandability, these ongoing efforts are classified according to the formalism applied. More precisely, ongoing proposals for service foundation are categorised into four classes, based on (1) Petri nets, (2) process algebras, (3) graph transformation and (4) temporal logic, as follows.

Petri-net-based service foundation: Since BPEL is the most widely adopted process-centric WS standard for orchestrating composite services, most if not all proposals based on Petri nets have focussed on how to analyse BEPL-based specifications. In [82] the authors propose a method of systematically translating back and forth between Workflow P/T Net (i.e. WFN) [100] and BPEL specifications. The first benefit of these two-sense translations is the formal analysis and verification of BPEL specifications using the simulation and verification capabilities of Petri nets (e.g. graph-reachability, P- and T-invariants). The second is that the verification of properties such as the absence of conflict / deadlock, termination and liveness has been addressed. In [84] using Petri nets, the authors checked properties such as compatibility (of a given service process to other partners) and usability (the existence of partners). Coloured Petri Nets (i.e. CPNets) [51] have also been used to formalise and validate BPEL specifications [107,108]. The advantage of CPNets is that more complex data-intensive, persistent and conversational orchestrations can be tackled. Another approach based on a predicate (architectural-based) Petri net variant called SAM has been proposed [44]. Since SAM integrates a linear temporal setting, more properties such as fairness and liveness can be addressed.

Process-algebra-based service foundation: Due to their concurrent and operational characteristics, process algebras such CSP and CCS have been rapidly adopted to express BPEL-based orchestration as well as WS-CDL-based choreography. In [76] and [79], for instance, the authors propose an adapted process calculus for reasoning on service composition using BPEL-like choreography.

Graph-transformation-based service foundation: Graph transformation formalism allows the modelling of any system structure as graphs and of their behaviours as graph-based rewrite rules. Its strength is thus in its graphically appealing descriptions. In [49], for instance, the authors propose graph-transformation to check formally the correspondence between customer requirements for services and what the (composite) service offers. This can help in the phase of service discovery, where non-functional properties such as qualities are considered.

Temporal-logic-based service foundation: Temporal logic formalisms have also been used to conceptualise WS applications because of their ability to verify any reactive system declaratively. In [93], colleagues from our group have proposed an adapted extension to the Interval Temporal Logic (ITL) to address formal composability in web services.

Also it is important to mention that in parallel to these formalisms, methodologies aiming to support most of the lifecycle while developing service-driven applications are starting to gain in maturity. For instance, Service-Component Architecture (SCA) [16], supported by graphical notations, aims to take service development to the level of business considerations. A first formalisation of SCA is put forward in [1] using architectural techniques. At IBM, the so-called SOMA (Service-Oriented Modelling and Architecture) methodology [10] is being promoted.

2.2 Business Rules and Interactions

Business rules (BRs) are defined as " *declarations of policy or conditions that must be satisfied to do business* " [65]. They play the role of governing operational decisions made inside the organisation. In other words, business rules specify the actions on the running particular business events, also it includes the 'state of being' and changes concerning (groups of) individuals, infrastructure, informational resources and business activities [53]. In dynamic business areas, the most frequent changes appear at the level of the business rules that specify the interaction of business entities.

This section discusses BRs in information systems, first distinguishing between intentional rules, operational rules and IS architecture rules, then showing how to structure these BRs according to the ECA mechanism. Finally, there is an overview of some recent approaches that integrate business rules in service technology and WB services in particular.

2.2.1 BRs in Information Systems

In recent decades, business rules have become an important element in information systems (IS), where activities are tackling with take care of rules which either prescribe a certain action or put condition to the set of possible actions. In this sense, BRs represent projections of organisational constraints and ways of working on their supporting ISs. Therefore, their collection, expression, structuring and organisation should be central within IS analysis. In [53] a repository structure for organising business rules for use within rule-intensive projects is proposed. In this repository, BRs are composed of intentional rules, operational rules and IS architecture rules, each of which is discussed below.

Intentional rules clarify all the laws, external principles with an explanation of how business could be conducted by its organisation; which is from the view point of business. These intentional rules use the same statements of our natural language. Their organisation in a repository environment concentrates on the apperance of their causal relationships (i.e. relationships with existing or future enterprise goals), information about the rule collection process itself (stakeholders that reveal them, versions, etc.) and details of their enforcement (date of enforcement, expiry date, status, etc.).

Operational rules are the view point of a business process perspective. It gives a description to each action on the business event appearance or valid states of organisational information entities. In general, operational rules are translations of informal intentional rules to formal rule statements deployed in accordance with a compatible rule language and repository schema. Moreover, reference is made to other enterprise knowledge concepts such as actors, activities, activity enablers, information objects and their attributes.

IS architecture rules are statements tested from an IS implementation view. They are introduced in an implementation-specific manner, i.e. in accordance with the implementation paradigm that has been selected for the IS under development (e.g. through the choice of a specific customisable rule engine, active databases, and so on). Normally, the identification, formation and organisation of IS architecture rules is covered by IS development approaches.

Generally ‘operational rule’ expressions are structured according to the ECA paradigm so they are used instead of intentional rules to describe the structure of information entities or prescribe actions on the occurrence of business events [54,106], as depicted in Figure 2.2.

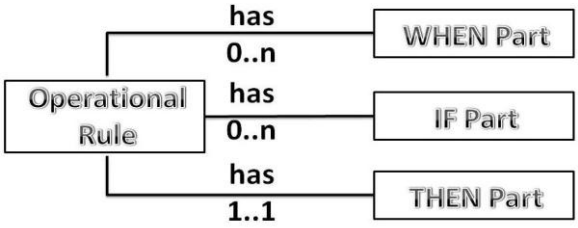


Figure 2.2: Operational rules with ECA, [54]

According to the ECA paradigm, *since an event occurs, and all the conditions hold, the specified actions must be performed*. An assured operational rule may consist of zero, one or more events (WHEN part), zero, one or more than one condition (IF part) and some actions

(THEN part). The WHEN part is used for saving the business event that triggers the execution of the rule. Events can be express as "*indivisible atomic units of action, with no duration, that occur in the real world, and affect at least one business entity*" [92].

2.2.2 Business Rules in Web Services

Given that all WS service standards, mainly with WSDL, BPEL4WS and WS-CDL [104], are by essence rigid, static and process-centric, they are far from being able to deal on their own with evolving rule-centric knowledge.

The first proposal aiming to benefit from the potential of business rules was made by Papazogoulou et al. [81], who propose a stepwise rule-driven methodology looking for make more better adaptability at runtime when BPEL-like standards used by composing WSs. This approach starts with a general specification of the target composition; and with the help of right classified business rules, the composition is then scheduled, constructed and finally executed.

The milestone consists of sorting in different warehouse business rules precise to WS composition. In addition to that, primary elements (eg. events, conditions and messages), this sorting includes specific rules handling the activity flows, the data required for their composition and the constraints to be respected. However, no formal verification or validation of the constructed composition is undertaken. Moreover, this approach copes only with functionality-driven rules and does not promote explicit rule-centric interactions.

Another innovative proposal was made by Dustdar et al. [87]. This work, is shifting from WSDL and its shortcoming which are passive and static to BRs by using reactive RuleML [88]. In this sense, rules can be discovered and composed like any services while being (internally) processed using logic-based engines such as Prolog [12]. The rules are thus considered as independent *agreements* to be invoked over the Web as services. The approach is automated with a supporting tool called ViDRE [78]. The weakness of this approach is that it neither addresses the conceptualisation level nor copes with dynamic composition of the WS rules to specific business process *activities*.

Apart from these proposals to bring BRs *directly* into service technology, the confluence of WSs and the Semantic Web [17] could be regarded as a radical alternative which permits the incorporation of *ontologically* interpreted knowledge into service technology. OWL-S [66]

and Daml-S [8] are among the leading languages in that category. Since the scope of this thesis is limited to explicit BRs, semantic WSs are beyond it.

2.3 Introduction to Architectural Techniques

As the complication in design and specification of software systems is raised, the demand on coarse-grained building blocks becomes essential. It has been addresseed in the area of software architecture and introduces high-level abstractions for representing the structure, behaviour and key properties of a software system. This section briefly discusses these architectural concepts and related architectural description languages.

2.3.1 Architectural Concepts

The software architecture of an application is the structure of that system, showing details of software components, their externally visible properties and the relationships between them, is also called connectors [15]. Software architecture contains: (1) elements description from which systems are built, (2) elements interactions inside them, (3) the patterns that guide their interactions and (4) patterns conditions [84]. These architectures are usually described using Architectural Description Languages (ADLs).

In a wider view, terms of a collection of components define specific system, and that could be done through the interactions among them (connectors) and their concrete interconnections (the set of components and connectors). An architectural description is depend on the building blocks which are thus (1) the *components*, (2) the *connectors* and (3) architectural *configurations*. An explicit specification take thier meaning from what ADL present. Modelled is so important to specify any sort information about architecture, at a minimum, interfaces of components. Figure 2.3 is a graphical illustration of architectural components. These architectural concepts are defined as follows.

• **Architectural Description Language:** Modelling a software system's conceptual architecture are presented in their features by using ADLs. The high-level structure of a system in terms of components and their interactions has been described by them, often referred to as structural models.

• **Components:** These are computations that have been executed to make certain the basic services are introduced inside the system. A component is an identity of computation or a

data store. For that, components are computations where states have been created [90]. In architecture, a component is small as a single procedure (e.g. meta-H procedures) or big as a huge application. A system is described in its physical (implementation) as an element or component along with their interactions. A system itself is also a component and systems can be composed of other systems.

• **Interface:** This is a set of interaction points acting as a bridge between the component itself and the outside world. A component interface in an ADL specifies those services (messages, operations and variables) that a component may provide or require in a given interaction.

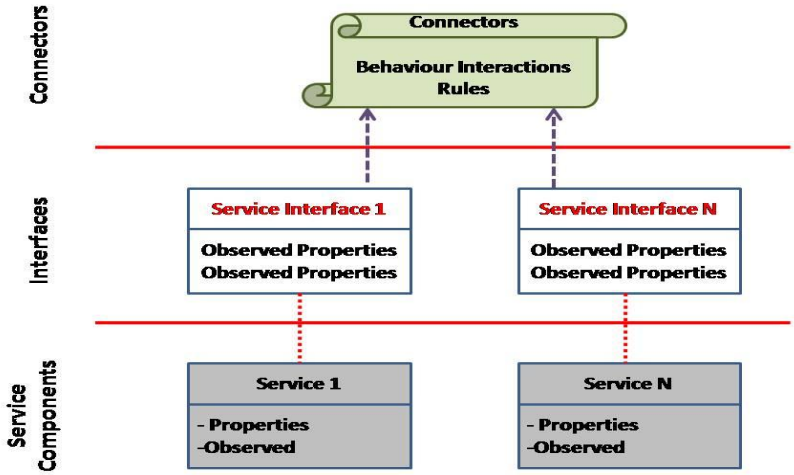


Figure 2.3: Graphical illustration of architectural ingredients[90]

• **Connectors:** These are a combination of the architectural and the rules, which is the architectural that building blocks used to model interactions among components and the rules that govern those interactions. They provide the glue for architectural designs and therefore deserve explicit modelling treatment. Connectors, in a runtime description, became between the communication and coordination activities within components. Good examples are good sample for simple forms of interaction, (eg..procedure call, and event broadcast). Also it could introduce difficult interactions, (eg.a client-server protocol or an SQL link between a database and an application).

• **Configuration:** An architectural structure is introduced in graphs that combine the connectors with components. It is important to specify if the right components are connected

and their interfaces match. Conectors allow communication and its own semantics in the proper behaviour.

• **Architectural style:** This determines how instances of component and connector types can be combined in a system or family of systems by using their vocabulary [91]. The way of choosing the good architectural style could be a success for the system during the design stage. Architectural evolution impacted by styles through reducing the changes an architect is allowed to make. Examples of styles include pipe and filter, client-server [91] and C2 [86].

Certain qualities have been achieved by the system design based on its composition from components and connectors. The architectural description can be introduced by the architecture of a software system. The next subsection discusses ADLs in more detail.

2.3.2 Architectural Descriptions Languages

An ADL for software applications does not focus on the implementation details of any specific source module but it gives more focus on the high-level structure of the overall application [101]. The aid to understanding of and communication about a software system is the core role of architectural descriptions. For instance, the simplicity of an ADL is too important, also the understandability and possibility of graphical syntax is needed.

Characterising software architectures usually has to present conceptual framework and a concrete syntax and that is done by ADLs [69]. Characteristics of the domain have been reflected by the conceptual framework for which the ADL is intended and/or the architectural style. Modelling architectures have proposed a number of ADLs within a particular domain and at the same time as general-purpose architecture modelling languages [70]. Examples are C2 [68], Darwin [63, 64], Rapide [98] and Wright [2]. Each of these languages gives specific distinguish abilities and all of them takecare of architectural design. For instance, an event-based style that has been uesd by the description of user interface systems is supported by C2, the analysis of distributed message-passing systems have been supported by Darwin, to make simulated solutions by architectural designs and then got the result of those simulations Rapide has to be used, while interactions between architectural components when its need formal specification and analysis Wright is the good solution for that.

2.3.3 Architectural Techniques in Software Evolution

The appearance of Internet technologies and the pace of reconfiguration that takes place in the commercial world, provide an important note that evolution must be taken care of at the architecture of systems level . From the other side, the competition that characterises the new economy leads to a volatility in the requirements that software systems have to satisfy.

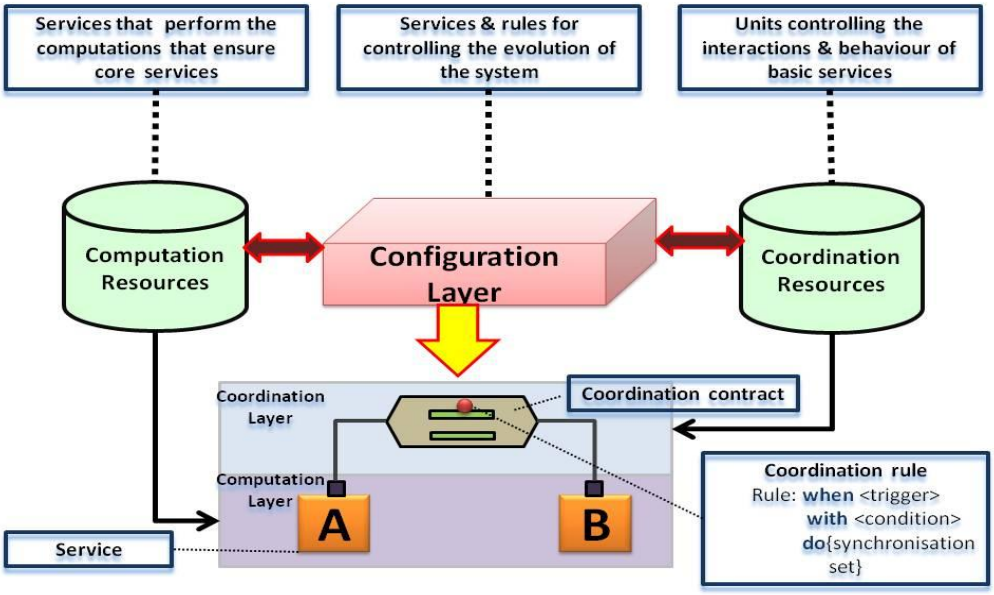


Figure 2.4: Three-layer architecture, [5]

A lot of repeated changes are happening not just in core entities level like a bank account but also at the business rules level or the protocols that specify how these essential entities are interacting as in customers acting with their accounts [5]. Therefore, a new methodological concept is important with its technology that supports it to help activation as first-class entities, this will take them to systems that are exoskeleton, in this way that they show their configuration structure explicitly [58].

These ingredients of methodology and technology are built upon three aspects of the development and deployment of any software system with a clear separation [6]: the computations introduced locally in its components, the coordination mechanisms through which global properties can emerge from those computations, and the configuration operations which ensure that the system will evolve according to given organisational or other policies (see Figure 2.4). To respond to any changing happened at the requirements

stage without any affect to fundamental objects that compose the system, the availability of aspects of coordination are needed.

Figure 2.4 illustrates and clarifies this three-level architectural methodology to develop intrinsically reconfigurable software. At the low-level, business entities have to be computational, represented here by symbols such A and B. Examples of such business entities are, for instance, accounts, customers, employees, etc. They represent real or computational independent entities from real-world entities of the application at-hand. At the middle-level, then interaction between such entities should be focus on. In this manner, the interaction are extract and external it. This facilitates its manipulation and superposition on the computational low-level of entities. Finally, the higher-level allows exactly the manipulation and the reconfiguration of the two already described levels (i.e. the entities and the interaction levels).

Of paramount important in this figure, with respect to adaptivity and reconfiguration, are thus the second and the third layer. With respect to the second layer, this approach adopts the event-condition-actions ECA paradigms. That is, the rules governing the explicit interactions between entities of the low computational-level are expressed in ECA-driven manner. They are then stored in respective rules repositories, which could be mainly a suitable database. We note that also low-level entities are stored in respective repositories to ensure persistency in the application. Finally, the higher-level permits to dynamically select the right rules, while composing or reconfiguring them accordingly.

2.4 Aspect-Oriented Techniques

Aspect-Oriented Programming (AOP) was first put forward by [57], in response to the limitations of the object-oriented programming in factoring out *cross-cutting* concerns (e.g. persistence, logging, security). The main concepts of AOP are *concerns*, *aspects* and *weaving*. In a broader sense of the word *concerns* and requirements are equal, which start from the first step in the level of design to the last one of this level with its details, and beginning of the next stage which is implementation.

An aspect, at the programming level is a modular unit that implements such a concern. Definition of an aspect consists of (a) behaviour or advice (code that must be executed), and (b) a specification or point-cut that expresses when, where and how to invoke the advice; is

conceptually defined as a predicate that evaluates over join-points.A join-point is a well-defined place in the structure or execution flow of a program where additional behaviour can be attached. Finally, weaving is the process of composing main functionality modules (normal components of any application) with aspects, so achieveing a working system.AOP provides language mechanisms to capture cross-cutting structure. In this manner, programming cross-cutting concerns will be more easy to create in a modular way and to achieve and gain the modularity improvments; while development and maintenance will be more simple to be coded especially as they take part in researchers' view to improve and help reduce the cost and time.

AOP thus allows cross-cutting concerns to be extracted from different code units (e.g. components, modules or classes) and externalised in so-called advices, as factorised encapsulated behavioural units ready to be inserted accordingly into specific positions in the units concerned. The positions where these advices have to be woven are called join-points, while the different ways of combining such advices before superposing them on respective units are denoted by point-cuts, a point-cut being the brief description of the set of the join-points where an aspect should be applied.

These main AOP mechanisms were first implemented in the AspectJ language [56]. Since then, different AOP languages have been introduced to address specific issues such as dynamic and / or compositional weaving mechanisms, strategies for advanced point-cuts, integration of reflection and componentisation.

2.4.1 Architectural Techniques and AOP

Nowdays, the principles of Aspect-Oriented Software Development (AOSD) have taken more consideration [43] with the main focus being on the systematic identification, modularisation, representation and composition of (often cross-cutting) concerns or requirements throughout the entire software development process.

To obtain the full benefits of aspects and make it used wider emerging the concepts of Component-Based Software Development (CBSD) to AOP was one the good ways to got this benefits [75]. This subsection illustrates how the principles of AOSD can be introduced into a component model, starting from the presentation of a fundamental component model.

Let us take as an example a simple banking system to be considered as shown in Figure 2.5. It contains three components; namely the CustomerClient component, BasicBanking component and AccessControl component. The first one deals with logging the customer into their account and doing their transactions. The second component introduces the core operations that help to manage the account. The last component is to authorise the customer to access their account, and also deals with authentication. Those components obtain their interface and need special requirements. Connectors are responsible for matching interfaces (graphically represented as dashed arrows).

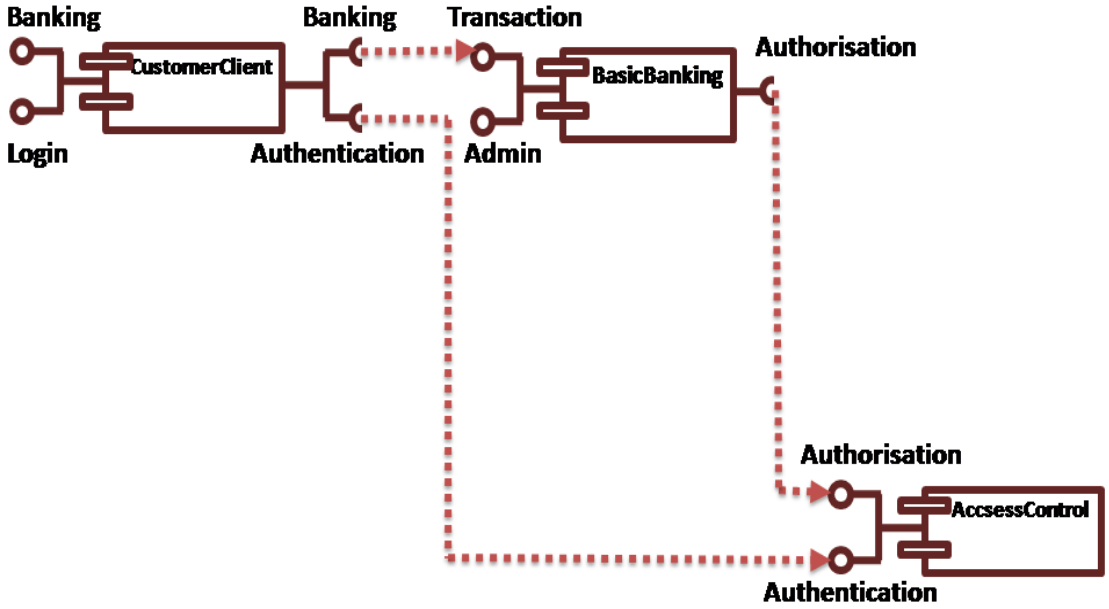


Figure 2.5: Component model of a banking system

Join-points, the point-cut language, advice, aspects and the weaving mechanism are considered when the concepts of AOP are mapping to a component model. Aspects and module have the same important because teh module has consist of point-cuts and the binding with the cross-cutting functionality.

Concerning aspects and the weaving mechanism, the aspect model can be incorporated into the component model in more than one way. In the following, two of these methods are described.

• **Aspect components** (incorporating aspects as components). In this method, the component absorbs the full behaviour of the aspect. That is, in addition to the introduce the functionality as required by the provided interfaces, depending on the functionality of the required

interfaces, an aspect component contain exported cross-cutting behaviour in way of provided and required interfaces. Because a point-cut is a predicate over join-points and a join-point is an entry on the interface of a component, a point-cut could be exported as a required interface of the aspect component. The semantics of such a required interface are that the aspect component will add cross-cutting behaviour to all the components connected to this required interface. The advice included in the aspect component may or may not be exported as normal functionality on a provided interface.

Going back to the example, the *AccessControl* component could be available as an aspect component (see Figure 2.6). It will weave functionality at the join-points as specified by its required interfaces, which are bound to the relevant interfaces. In this case, authentication functionality will be woven in when a customer uses the client and authorisation will be woven in before all banking operations.

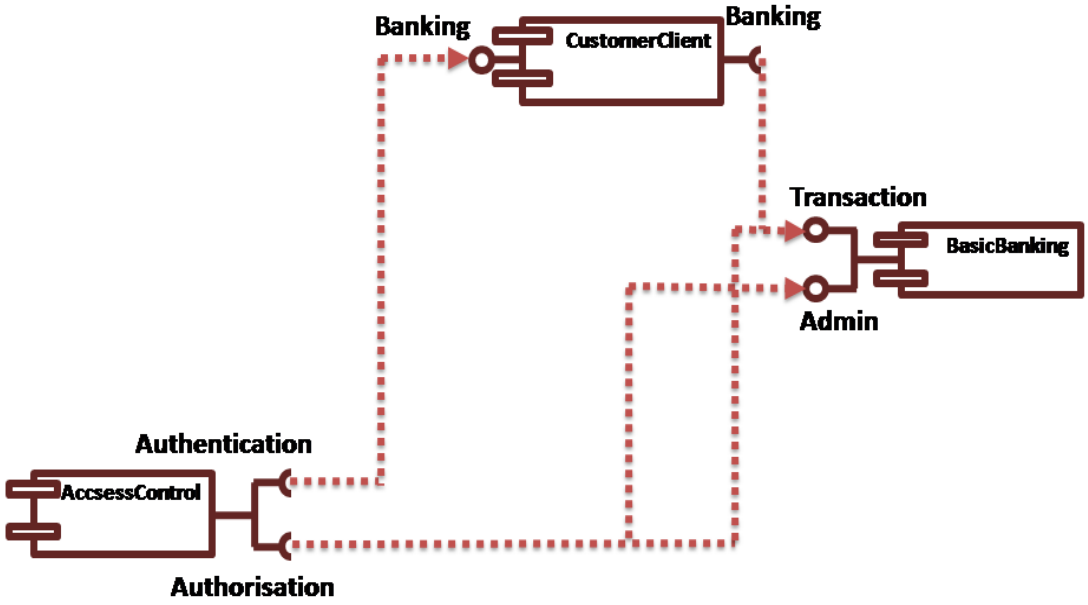


Figure 2.6: The banking example using aspect components

The difference between the two conceptions, that is, without aspect and with aspect as well as the benefits to be gained from the later, can more highlighted on the basis of this example as follows:

In the conception at Figure 2.5, that is, without benefiting from the aspect mechanisms, we have two (duplicated) functionally-similar components, namely the CustomerClient and the AccessControl. Indeed, both components involve the authorisation and authentication to access basic functionalities of the banking and performing transactions.

Such duplication of components is well-known as cross-cutting concerns, from the object-oriented modelling, and represents serious source of conflict, ambiguities and inconsistencies. Indeed, assume we change the authorisation at the CustomerClient. This will directly create a conflict and inconsistency with the second similar component, namely the AccessControl. With respect to realistic applications, it may easily hand dozen if not hundreds of such functionality-similar components, to implement among others security, performance, management issues for different components. The results will then be thousand of incompatible codes scattered and tangled to different components.

In Figure 2.6 aims at circumventing this unfortunate situation, while benefiting from the aspect-oriented mechanisms. More precisely, first, all functionally-similar components have to be bring together, into one component, it call aspectual component. That is, instead of the CustomerClient and the AccessControl, just one component is conceiving that brings together all functionalities, concerning the authentication and identification. In this sense, the CustomerClient is now concerned only with the banking functionalities. All cross-cutting functionalities concerning logins have to be externalised, authentication and identification to the AccesControl component, which becomes now at the aspectual-level.

By merging all cross-cutting concerns dealing with authentication, identification and logins into a single *aspectual* component, namely AccessControl aspectual component, the gained benefits are manifold. First, no need to worry about any source of inconsistency or conflict. Indeed, there are no duplicate authorisation or authentication functionalities, but instead just one unit with different possible arrangement, as point-cuts. Second, the wished functionalities can dynamically at the right-time be weaved on the right component. In this sense, the authentication functionalities are woven, through respective join-points, on the CustomerClient component; whereas, the authorisation functionalities are woven into the BasicBanking component. Last but not least, these functionalities can be now updated and manipulated independently from the components at the aspectual-level, which allows high-flexibility and adaptively while staying consistent.

• **Aspect connectors** (incorporating aspects as connectors). Under this option, most of the aspectual behaviour will be absorbed by the connector. The definition of the point-cuts and the binding of those point-cuts with the device will be specified in a connector, while the device functionality is available as a normal service on the interface of a component. Besides

normal connectors, binding provided interfaces to the matching required ones, there are also aspect connectors, which are able to bind some provided functionality before, after or around a set of join-points.

Aspect connectors can offer more flexibility than aspect components. All components can be developed as usual; the developer does not have to choose whether to develop a normal component or an aspect component. Only at composition time will it be determined whether a component is used as an aspect component or not. This will make the component more reusable, because it has not been decided whether the functionality offered by the component is cross-cutting or not. Aspect connectors can be considered to function as composition descriptors, configuring the components which are to interact with each other.

At the requirement level, several proposals have been made for extensions of UML diagrams to handle (aspect-oriented) cross-cutting concerns [45, 102]. At the architectural level [24], ADLs have been enriched with aspect-oriented mechanisms. For instance, in [14, 46], besides usual components and connectors, the concepts of aspectual components and more importantly aspectual connectors have been proposed to transparently and flexibly cope with cross-cutting concerns such as distribution and security at the architectural level. It is also important to mention the serious attempts such as by [31, 42, 60] to interpret AOP rigorously at the logic level, mainly using the PROLOG logic programming language.

Despite these proposals, this thesis argues that much work remains to be done in order to establish a satisfactory and rigorous handling of cross-cutting concerns at these early *requirement-architectural* levels. More specifically, it argues that the following issues remain largely unexplored. First, the abovementioned ADL extensions with aspect orientation put exclusive emphasis on structural features while disadvantaging important behavioural features (e.g. connector glue behaviour). Secondly, the formal underpinning of such proposals with highly declarative semantics is still lacking, thus hindering any realistic rapid prototyping and/or sound verification. Finally, adaptability (as advices) is addressed only at the design-time, so that the desired runtime adaptability is not supported.

2.4.2 AOP in Web Services

As aspect-oriented concepts and techniques represent the best available software engineering mechanisms to address the separation of concerns and adaptability, several ongoing

proposals benefit from these AOP capabilities in developing agile service-driven applications.

A first interesting proposal in this sense is that made by Charfi et al. [22], with the main aim of bringing more agility and modularity to the BPEL language, by enriching it with an extra aspectual level. The resulting new language, named AO4BPEL [23], allows the externalising of cross-cutting concerns such as security and data handling by separately codifying them as XML schemas and then (statically) weaving them into BPEL activities. AO4BPEL supports two kinds of join-point: process-level and interpretation-level join-points. The former correspond to the execution of an activity and the latter to internal points during the capturing of an activity, e.g. the point where a SOAP message corresponding to an invoke activity has to be created. AO4BPEL uses the XML-Query language XPath as its point-cut language. This approach has been abstracted recently to fit any workflow language (i.e. going beyond BPEL) [21]. The approach introduced by Erradi et al. [39, 40] also adopts aspect orientation and is specifically devoted to policies and QoS concerns. It is not only adapted to BPEL and it addresses the runtime weaving of different policies on running services. In [41] an environment called MASC has been developed on .Net-based WSs. In the same line, Finkelstein et al. propose in [33] a generic aspect-oriented language which can be applied to the dynamic weaving of behavioural advices in BPEL code. The approach uses a Java-based supporting tool called SmartTools.

Another aspect-driven approach to WS appeared most recently in [77, 89], placing emphasis on the adaptability of business protocols while composing web services. Indeed, syntactical and semantic mismatches often both occur while invoking and composing complex services. For instance, a customer opts for some activity ordering which the provider cannot directly deliver (e.g. the user wants goods shipment first, whereas the provider requires the payment first). To resolve such frequent mismatches dynamically, the authors adopt a higher aspectual level, where dynamic ordering can be reached on the fly.

2.5 Maude

As noted in the introduction, the approach adopted in the present research is underpinned by rewriting logic and the Maude language [27]. Maude is a high-level language and a high-performance system supporting both functional and object-oriented specifications and programming for a wide range of applications (more details in Appendix C). Maude has

been influenced in important ways by the OBJ3 functional language [48]. In particular, Maude's underlying equational logic sublanguage, namely, membership equational logic, extends OBJ3's order-sorted equational logic [47]. The main features and characteristics of Maude are rewriting logic-based semantics and reflection.

Maude modules are rewriting theories, while computation with such modules corresponds to efficient deduction by rewriting [35]. There are three kinds of modules in Maude: the functional module (**fmod**), the system module (**mod**) and the object-oriented module (**omod**).

2.6 Service Technology and Foundation: Comparative Study

The purpose of this paragraph is to present a general assessment of the above approaches with respect to the approach proposed in this thesis. Furthermore, a more elaborative attention will be given to two approaches forwarded in STRL laboratory, namely the one based on ITL logic [94] and the one based on Grammar-objects [9]. Nevertheless, due to several objective reasons, it is not an exhaustive comparison. First, all the related summarised works are based on still ongoing research ideas and are far from being widely adopted at the industrial-level. Second, most if not all the above approaches emphasised more the modelling and certification, whereas this work has been particularly focussing on dynamic adaptively. Last but not least, the formal foundation adopted by these work are quite different from the algebraic and rewriting-logic based we adopted in this thesis.

With respect to Petri nets-based approaches to service foundation as well as graph-transformation-based one, the notion of service behaviour is mainly restricted to the causality between service operations (i.e. business protocols). That is to say, these approaches represent abstraction-level, where the activities are conceived just as black-boxes. In the proposed approach in this thesis, a further externalises at the adaptability or aspectual-level, with going inside any activity and models its behaviour as ECA-driven rules. In this sense, these approaches could be benefited from, before going inside the activities to check the correctness of the global business process, but in a static manner.

The process algebras-based approaches focus also of the causality of service operations and their composition but at a more concrete and operational level. Our adaptively mechanisms based on Maude reflection strategies could be regarded as dynamic and adaptive

composition of process business activities. In this sense, a mapping between the strategies and the corresponding process in terms of algebras could be investigated as further work to this thesis.

In the following two paragraphs, we address in some detail the two mentioned approaches developed in STRL Laboratory and compare them with the forwarded results of this thesis. Again, worth-mentioning that due to our focus on dynamic adaptively with aspect techniques and the adoption of the rewriting setting as formalisation, it would be very hard to come up with convergent and exhaustive comparison to these approaches.

2.6.1. Comparison with Arsanjani's Approach

This approach developed in the PhD thesis of A.Arsanjani consists in a general-purpose methodology called GOOD for developing complex component- and service-oriented systems, while covering all their life-cycle and emphasising their dynamic reconfiguration. In some detail the main milestones of the approach could summarised as follows:

The methodology starts with description of main goals and sub-goals as well as business architectures, underlying the main application functionalities to cope with.

Once this first phase is achieved, the methodology proposes to tackle non-functional features by defining different component- and service-oriented software architecture. At this level, domain-specific languages can be adopted, depending on the peculiarities of the application at-hand.

Now to *dynamically* compose such service-oriented components in re-configurable way, the approach proposes different mechanisms including: (1) Grammar-oriented mechanisms, to control and refine actions to be performed starting from the goals to fine-grained and low-level activities, dealing with the storage and menus. The next (2) step in this essential phase is to steer the composition with the supports of the so-called *manners*. Manners are context-aware externalisation of component behaviour, allowing non-intrusive changes to be applied on component’s flow, rules.

The Business Grammars that steer the re-configurability with manners represent the heart of the approach, the so-called Controller-layer. That is the approach reposes on the MVC (i.e. Model-View-Controller) paradigm. The models are the concrete technologies for services

such as EJB, .NET. The View layer represents the interfaces of the application, such as Menus and Windows. We further note that the manners are usually refined into UML activity-diagrams.

The methodology is supported by a compliant tool that starts from the definitions of goals, define the business architectures, the service architectures, the manners and end with the implementation on the models and the building of the menus and related user's interfaces. The approach has been applied to the so-called E-Bazzar online Shopping system.

With respect to this thesis approach, there are several similarities and complementarities than differences. The main objective that both approachs strive for consists in developing adaptive and reconfigurable service-oriented applications, though this approach concerns also component-oriented approaches. The software-engineering techniques adopted are a little different and complementarily.

In our approach, although the definitions of goals and objectives of the expected service-oriented application or alliance has been started, the definitions of sub-goals and involved entities and components have not been elaborate more on. Indeed, an activity-based approach has been followed, that is, with respect to an identified business process, each activities has been looked insided to find out what are the governing rules for its behaviour.

In this approach the manners could be regarded as separate aspect-oriented mechanisms for controlling the adaptivity and re-configuration, through the author does not mention the AOP concepts but emphasise the externalisation of the manners from components and services. The main advantage over our approach is that this approach tackles also the service-oriented technological-level; which has been mentioned as future work in the conclusion. Another difference with our approach is the granularity of adopting business rules. In our case, the ECA-driven rules are defined at the activity-level, whereas in this approach the rules seem to be used at the business process and service coarse-grained level.

To conclude this approach is very promising and can be benefit from it much for future work. Among others, it can be benefits from the refinement of the goals and sub-goals to define our business processes in a more disciplined manner. The refinement of business architecture to service-oriented architectures could also be of great interest to derive concrete technology-based service-oriented solution from our conceptual model. On other side, in

would be interesting to upgrade the manners at the aspectual-level and adopt our (un-) weaving mechanism and formal validation in this approach.

2.6.2. Comparison with Solanki's Approach

This approach proposes to enrich interface specification of services with properties that enable reasoning about ongoing behaviour. The approach also provides a methodology for validation of those properties at runtime. In this sense, the approach endow service-oriented applications with a sound model of computation for services as well as a wide spectrum language that allow design and reasoning about the properties of a service and its composition at an abstract level and within a unified formal framework. More precisely, the approach developed in this thesis put forwards the following research results:

A computational model that defines the set of behaviours associated with a reactive service. This model is based on the ITL temporal logic framework.

A wide-spectrum language based on that computational model, with semantics in the underlying logical framework of ITL, to describe service-oriented systems. The so-called ASDL language is further endowed with a compositional capabilities and runtime validation of the composition of services.

An architecture and tool support to enrich ontological frameworks for services with the compositional specification. More specifically, a Web-service semantics language, called OntoITL, is developed while extending existing such as TeSCO-S and SWRL.

This property-oriented fundamental approach more complements this thesis approach, rather than overlap or being at cross-purposes. Indeed, whereas this thesis has been focussing on validation through executable specification, it emphasis the properties verification. Common to both approaches is the enrichment of service interfaces with behavioural features, in terms of constraints and rules. Moreover, this approach explicitly distinguish and externalise such ECA-driven behavioural from the service interfaces, so that it facilitate the runtime adaptability. Nevertheless, it could be forward a semantics-web service language that precisely reflects these ECA-driven rules, as this approach proposes with OntoITL.

2.7 Summary

This chapter has overviewed the main service-oriented techniques as well as all the software engineering concepts and mechanisms that will be recapitulated in following the envisioned approach to developing service-oriented applications that are both runtime adaptive and knowledge intensive. These required software engineering mechanisms include architectural techniques, aspect-oriented mechanisms and business rules for adapting applications. The chapter has also summarised the most important recent work and proposals aimed at bringing more adaptivity and rigour to web services.

It began by recalling all elements of service technology, such as standard web languages and service-oriented architecture, then surveyed the concept of business rules as an essential means of adapting information systems in general. It next described the most recent practical proposals for relating business rules to web services, then listed some formal approaches to WS standards and the service paradigm in general. Basic architectural concepts such components, connectors and configurations were reviewed, as were all of the essential concepts underlying aspect-oriented programming as a new paradigm for the separation of concerns and adaptability. The next section sketched ways of relating architectural techniques and aspect-oriented mechanisms. Finally, there was a presentation of recent proposals and ongoing work to allow web services to benefit from the potential of aspect-oriented mechanisms, with the aim of enhancing adaptability and separation of concerns.

Addressing all these varying ingredients and preliminaries to our envisioned approach shows again how complex and challenging is the dynamic adaptability problem within the service-oriented paradigm. Indeed, without business rules, adaptability would be reduced to mere rearrangement of black-box activities. Then, architectural techniques would be unable to tackle the problem of adaptability at the heart of service orientation, namely inter-service interactions and composition. Lastly, as will be explained in this thesis, without aspect orientation it would be difficult, if not impossible, to discuss runtime adaptability in a transparent and conceptual manner. Finally, to bring a precise and well founded semantics to any proposed approach requires formal techniques, without which no formal validation or verification could be achieved.

Chapter 3

A Rule-centric Architectural Conceptualisation and its Aspectual Extension

This chapter proposes an intuitive and descriptive event-driven architectural conceptualisation for agile and knowledge-intensive service-oriented applications. It should be recalled that this thesis reports work mainly at the conceptual level, since several severe challenges remain to be overcome before service orientation delivers all the promises for the reliable and flexible development of complex and agile cross-organisation applications. The approach envisioned here takes the following features as conceptual milestones.

In order to tackle adaptability, the approach adopted is to externalise the business logic from different services, mainly in terms of event-driven business rules governing any business activity within or between services. In other words, the aim is to overcome the behavioural limitations of WS interfacing standards such as WSDL, which can cope only with messages and operations in a given service [95]. This observation also applies to service composition languages, particularly BPEL for orchestration and WS-CDL for choreography, neither of which addresses behavioural features at the level of business logic (in terms of business rules) but only at the process-level (activity flow) [96]. To overcome this severe limitation in knowledge-intensive adaptability, it is proposed to externalise the behavioural features of any service as well as possible inter-service compositions, by adapting the evolution of associated ECA-based business rules.

To bridge the gap between business rules and service concepts (e.g. service interfaces, composition and interaction), such business rules are leveraged towards ECA-based architectural interconnections. This construction ensures *design-time* adaptability and evolution, since the rules governing service interactions become separated from the services and their interfaces so that they can then be independently manipulated (e.g. created, updated, deleted and evolved). Roughly speaking, in the envisioned service model of ECA-driven architectural interactions, the interaction interfaces will play the role of service interfaces, whereas the rules governing the interconnections will play the role of the service behavioural composition.

In order to facilitate the dynamic composition / interaction and discovery of adaptive and rule-intensive service-oriented applications, this ECA-driven conceptual model of architectural interactions will be further incrementally improved towards inherently supporting *runtime* adaptability. For that purpose, as outlined in the second chapter, advanced aspect-oriented mechanisms and concepts will be recapitulated [56]. As subsequently explained in detail, appropriate steps are proposed for the runtime interception of events, passing them to the composition level, executing associated rules at that interaction-level and dynamically weaving such behaviour into respective services.

The proposed approach thus begins by informally describing the behaviour of any business activity as adapted event-driven business rules. It then promotes these rules as rule-centric and service-oriented transient architectural interactions. Finally, they are gradually leveraged towards dynamic interactions by benefiting from aspect-oriented mechanisms.

The main purpose of this chapter is thus to delineate an aspect-oriented architectural approach that is independent of any service technology and infrastructural restraints. The approach is further required to promote design-time and runtime adaptation at the intra- and inter-service composition levels. This conceptual approach will be explained and illustrated using a simplified case study of service-oriented banking.

The remainder of these chapter first sets out the main steps and conceptual milestones of the approach, before detailing its conceptual architectural level and illustrating it with a simplified banking application. Finally, it considers how to integrate aspect-oriented mechanisms into this ECA-based architecture so that it becomes dynamically adaptable.

3.1 Milestones and Steps of the Approach

The general working architecture of the proposed approach is depicted in Figure 3.1, which shows that it comprises three main steps, explained below. As with any proposed conceptual approach, some knowledge of the application domain is assumed. In this case, for the successful use of the approach, the following assumptions are made. To ensure the reliable development of any adaptive rule-intensive service-oriented application, an informal description of it should be given. More precisely, during the requirements elicitation phase, the main goals and objectives of any opportunistic or strategic alliance of cross-organisational services should be specified. Furthermore, as is usual in the elicitation of

(distributed cross-organisational) information systems requirements, such objectives should be expressed in terms of broad business processes with their informal composed business activities. As explained in section two of the second chapter, intentional business rules governing the application concerned must also be (fully or partially) described at that informal level.

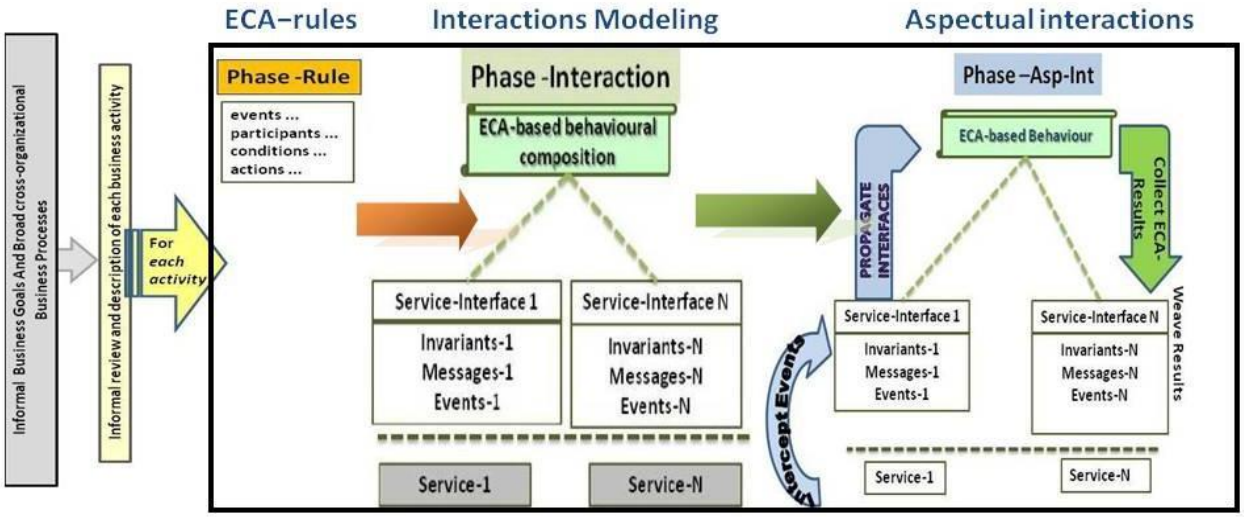


Figure 3.1: Stepwise abstract ECA-based aspectual architectural model

ECA-driven Business Rules for Activities (Phase-Rule): The first step consists mainly in understanding behaviourally each of the activities involved in given cross-organisational business processes of the application at hand. More specifically, it is proposed to describe the behavioural features of each business activity in terms of governing event-driven business rules. For that purpose, an informal ECA rule pattern is introduced, as being convenient and understandable. It begins by listing the events that trigger any business activity, then the constraints to be observed and finally the actions which are required to be performed. It is important to note that since composition is the driving force in the service-oriented paradigm, special emphasis is placed on interactions in formulating the governing business rules. This requires the enumeration of the business entities involved in any ECA rule governing a given activity. These participant entities will in the next architectural phase be regarded as services, from which are selected the interfaces required to express any rule.

Rule-centric Architectural Interconnections (Phase-Int): From the informal ECA-based behavioural description of any business activity in a given service-oriented application, the purpose of this phase is to shift to a more *disciplined* yet interaction-centric conceptual modelling. A further aim is to bridge the gap between the business level and logic and the

corresponding service-oriented level, where service interfaces and their behavioural compositions should occupy the foreground. Towards that ambitious aim, as outlined in the first and second chapters, the proposal is for architectural techniques and their transient architectural connectors. Indeed, as already noted, architectural techniques permit the externalising of interactions as first-class connectors. Such connectors involve service interfaces and behavioural glues, reflecting the logic of interconnection or of the service composition. To stay compliant with the ECA-driven rules, it is proposed to express connector behaviours in terms of a precise ECA-driven pattern.

Aspect-oriented Architectural Connectors (Phase-AspInt): The above ECA-driven conceptual model of architectural service interactions is intrinsically able to deal with design-time *adaptability*. That is, it is possible at any time during the design to add, update or remove a given service (interaction) behaviour-as-ECA rule by stopping the running application. However, what is not possible is runtime adaptability, that is, changing or evolving any service composition behaviour-as-ECA rules without stopping, replacing or compromising the distribution of the running service components involved and their interfaces. As just emphasised, such runtime adaptability is essential for the service-oriented paradigm. Indeed, first, requestors can decide to change their (behavioural) requirements while performing a given business activity. A typical example is that when a traveller wants to book a ticket, (s)he should be able to change optional requirements such as car rental or additional luggage. Second, on the provider side, it is more important to attract customers by giving them dynamic discounts depending for instance on customer profile, specific destination or seasonal periods (Christmas, summer). Third, while processing any complex and long-running service composition, some services may become dynamically disconnected, unavailable or unsatisfactory. In such cases, the application must be dynamically composed with other available and behaviourally equivalent ones. All these fundamental and omnipresent issues in service-oriented computing can be addressed with runtime adaptability and only with it.

To leverage the design-time service adaptability of the ECA-based architectural conceptual model, it is necessary to recapitulate advanced ideas from aspect-oriented mechanisms. More specifically, a five-step approach is proposed to leverage the above ECA-based architectural connectors in order to: (1) dynamically intercept events triggering any behavioural interactions; (2-3) propagate required messages and properties from service components

through their service interfaces to the interaction level; (4) invoke the interaction rules applicable to the service instances concerned at that composition level; (5) weave the resulting behaviour into participating running service components via their interfaces. It is worth pointing out again that this dynamic adaptability is proposed at the abstract descriptive *modelling* level, without mentioning any specific operational details. The next chapter contains proposals for using rewriting logic and Maude as a suitable alternative for formalising, performing and validating this conceptual approach.

3.2 Behavioural Services and Interactions with Event-driven Business Rules

As explained above, in order to address adaptability in service-oriented applications, the first step in the envisioned approach consists in expressing the behaviour of any business activity involved in terms of adapted event-driven business rules. That is, with the aim of developing reliable yet adaptive complex service-oriented applications [83], it is primarily proposed to focus on the intuitive business level. More precisely, given any application objectives and roughly defined (cross-organisational) business processes, as well as any associated intentional rules governing them, the proposal is to describe such intentional rules in a more disciplined and operational manner. This requires any informal intentional rule governing a given business activity to be replaced by a more operational one following the widely-recognised Event-Condition-Action pattern. It should be added that since the service-oriented architecture and enabling Web services are in essence event-driven, it was found that the ECA pattern was the most suitable way to capture business activity behaviour in the service paradigm. Besides, as already emphasised, since the service-oriented paradigm is marked by the composition and interaction of services, particular emphasis is placed here on interactions; that is, at this intuitive level, there is a description of the participant service entities involved in any business activity and its rules. The following two sub-sections first detail the ECA pattern adopted, then illustrate it with a simplified banking example.

3.2.1 ECA-driven Interaction Pattern for Activity Behaviour

The ECA business rules pattern [11,106] is thus proposed to govern the behaviour of any business activity. More precisely, the different clauses composing any business rule governing the behaviour of a business activity can be detailed as follows.

Activity and rule name: Each ECA rule governing a business activity begins by recalling the activity. A representative and expressive name can also be given to the corresponding business rule. Finally, when the possibility of ambiguity is high, this is addressed by also specifying the name of the business process associated with the activity and rule.

Participating entities: To emphasise the interaction-centricity of most service-oriented behaviour at the business activity level, the participating service and business entities are identified. However, at this abstract level, no detail is given as to what is required of such participating entities or their roles. This will be the focus of the next architectural level modelled.

Triggering events: Since an event-driven vision characterises most service-oriented business applications, it is necessary to specify the events governing such business rules. It is worth noting that a business activity trigger may range from a simple single event to complex and composite events (using operators of different kinds such as parallel, sequence and choice).

Constraints: The third essential element of any ECA-driven business rule is the constraints and conditions to be observed by the rule. These have to be described at this business level and checked before the rule is enabled, allowing such behavioural interaction between participating entities. These constraints represent the business logic to be extracted and externalised from the participating entities / services at the interaction level. In this way, they can be manipulated independently.

Actions: Finally, it is necessary to specify the actions to be performed as effects of any ECA-driven interaction behaviour. That is, once the triggering events occur and all associated constraints hold, then the related actions are to be enforced on different participating entities.

As a summary of this proposed pattern of ECA-driven rules, the box below contains a description of all of the rules to be applied in terms of the above-described clauses.

INTER-RULE: ECA-rule name (plus business activity name and process name).

PARTICIPANTS: Names of different participating entities.

EVENTS: Triggering events to invoke the rule.

CONSTRAINTS: Description of the constraints and conditions to be applied to the rule.

ACTIONS: Different actions as effects of the rule.

3.2.2 Simplified Banking Example of the ECA-driven Pattern

In order to be competitive, banking systems frequently offer attractive packages to their privileged customers. These may range from basic agreed contracts (e.g. different formulas for withdrawal or transfer of money) to sophisticated and complex offers (staged housing loans, mortgages, etc.) depending on customer profiles (e.g. assets, trust, experience).

That is to say, even for a basic withdrawal it is no longer acceptable to hide the business logic statically inside the service components, as was done, for instance, in the Maude illustration in appendix C. Indeed, inheritance is not a good way of changing guards and conditions on methods, in order to model different situations. First, inheritance views objects as white boxes in the sense that adaptations such as changes to guards are performed on the internal structure of the objects, which from the evolutionary point of view is not desirable. Second, from a business point of view, the adaptations that make sense may be required on classes other than the ones in which the restrictions were implemented. In the present example, this is the case when it is the type of client, and not the type of account, that determines the nature of the guard which applies to withdrawals. The guard will be applied to debits and specialisation to accounts, because in the traditional clientship's mode of interaction, the code is placed on the supplier's class.

Therefore, it makes more sense for business requirements of this sort to be modelled outside the services which model the basic business entities, because they represent aspects of the domain that are subject to frequent changes (evolution). The present proposal is that guards like the one discussed above should be modelled as ECA-driven interactions, which can be established *between customers and account services*. That is to say, a withdrawal must be

regarded as an agreement between the customer and his/her account services, which directly leads to more transparent, flexible and adapted withdrawal.

More precisely, following the above generic rule-centric pattern for interactions, the following ECA-interaction rule, called standard withdrawal (Std-Wdr) is proposed. It consists simply in externalising the withdrawal constraint (i.e. balance > amount) from the account service component to the interaction level.

The rule is first described as an intuitive intentional rule, which may be part of the initial requirements, as follows.

Rules (withdrawals4customer): Any customer possessing an account service in a bank can withdraw money (using an ATM for instance) in an amount not exceeding an agreed threshold, upon acceptance of the payment of a charge. Privileged customers can be assigned credit to withdraw amounts larger than their account balances.

Under the ECA pattern with the explicit mention of the participant business entities, this informal description can be refined. For instance, the first type of withdrawal takes the form:

INTER-RULE: Standard Withdraw.

PARTNERS: Customer and Account entities.

EVENT: The customer requests withdrawal of an amount from an account service.

CONSTRAINTS: The customer owns the account and the requested amount is not greater than the available balance.

ACTIONS: The balance is debited with the requested amount via a debit action.

After the externalisation of the event-driven business rule which determines the conditions under which withdrawals can be made, its evolution can be supported by defining new ECA-

interaction rules, between the customer and his/her account, which may replace or augment the old ones. For instance, consider the following ECA-interaction rule, namely VIP withdrawal, which allows the relaxing of the functionality of withdrawal so that an account may be overdrawn through giving a property credits:

INTER-RULE: Withdraw with credit (VIP).

PARTNERS: Customer (with specific credit) and Account entities.

EVENT: The customer requests withdrawal of an amount from an account.

CONSTRAINTS: The customer owns the account and the requested amount is not greater than the available balance plus the credit assigned to that customer.

ACTIONS: The balance is debited with the requested amount via a debit action.

3.3 ECA-driven Architectural Interconnections

The next important phase aims to bridge the gap between the business level discussed above and a more disciplined conceptual level, where main service-oriented mechanisms are abstractly present, particularly service interfaces, service behaviour and service composition. Furthermore, as this research is concerned with behavioural service adaptability and rigour, this conceptual level should promote these two main preoccupations. As outlined in the second chapter, behaviour-intensive architectural techniques, characterised in particular by transient interconnections, seem to be the most suitable for this phase and its challenges. The aim is thus to use architectural techniques with their transient connectors to leverage smoothly the level of rigour and discipline of the above ECA-driven business rules, while enhancing their intrinsic interaction-centricity, agility and evolution.

The following subsection first proposes an adapted and generic ECA-driven architectural pattern for the conceptual modelling of the envisioned service. It then outlines how the abovementioned intuitive ECA rules are smoothly leveraged to the disciplined architectural level. The next subsection illustrates this conceptual model of service architecture by applying it to the above ECA-driven banking rules for cash withdrawal.

3.3.1 ECA-driven Pattern for architectural Service Interactions

Intuitively speaking, given an interactive ECA-driven business rule governing the behaviour of a business activity in a service-oriented business process, it is proposed to shift it towards a corresponding architectural connector through the following steps.

Depending on the rule elements (e.g. events, attributes and messages) required from different service partners, each partner or business entity name involved is transformed into a *service interface*. In the terminology of architectural techniques, this corresponds to a connector role. That is, for any partner, its service interface is precisely defined as composed of all messages, events and / or properties required by the rule to express the intended behaviour at the interaction level.

Still depending on the rule, specifically the constraint part, additional messages, attributes, constants and invariants can be defined as part of the interaction itself. In such a case, the interaction could be considered later as an independent third-party service; this could be located either on the provider side or independently considered to govern cooperation between the services involved.

The rule itself is finally captured in a more precise manner, but the ECA-driven paradigm is retained in describing the interconnection behaviour. In terms of architectural terminology the rule corresponds to the connector glue, whereas in terms of the service-oriented paradigm, this glue reflects the behavioural composition of services via their interfaces. The precise description of the rule, as detailed below, promotes the aforementioned ECA-driven intuitive pattern and utilises information on the service interfaces to leverage its expressiveness.

More precisely, the model of ECA-driven service architectural interactions which is proposed to make ECA-based business rules more disciplined can be illustrated by Figure 3.2. First, under the keyword *ECA-Interaction* is the name of the rule-based service architectural interaction. Next, some names are defined as instances of the entities involved under the keyword *participants*. It is important to point out that any such ECA-driven architectural interaction is graphically represented on the right-hand side. In this graphical representation, the name of the behavioural interaction is first placed within a contract-like graphical icon. Then, for each required interface, a graphical box within it is used to describe its name accompanied by all required events, messages and attributes. A dotted line then

connects the contract icon to each of the interfaces. Any condition that must always hold is designated by the keyword *invariant*. The case is also considered where besides the interface information, proper constants / attributes are required at the interaction level. These are described after the keywords *constants*, *attributes* or *operations*, depending on the associated case.

The ECA-driven interaction architectural rule itself begins with the keyword *interaction rule*, where the name given has to be the same as those of the selected ECA-driven intuitive rule, then the events triggering the rule follow the keyword *at-trigger*. The constraints to be checked to allow the application of the rule are specified after the keyword *under*. Finally, the actions to be performed when the triggering events and the constraints hold are specified under the keyword *acting*.

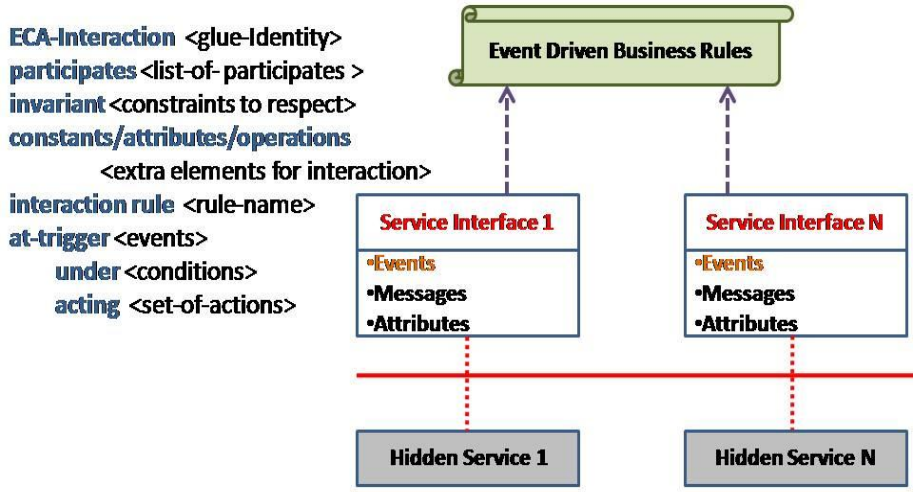


Figure 3.2: ECA rule-centric architectural interactions

3.3.2 The ECA Architectural Pattern Applied to Banking

This subsection reconsiders the two intuitive ECA-driven rules, governing standard and the VIP withdrawal, from the previous phase. That is, respecting the above guidelines and straightforward modelling translations, the corresponding disciplined ECA-driven architectural interactions could be constructed as follows.

As depicted in Figure 3.3, the ECA-driven architectural interaction for standard withdrawal is first considered. The left-hand side of the figure shows that beyond defining the customer and account entities as service participants, the required events, messages and attributes are also now specified. That is, the customer service is required to publish the event *withdraw (M)*, where *M* is the amount of money to be withdrawn. The account service must then give

(1) the *balance* of money currently in the account and (2) the basic operations in order to *debit* any account with an amount of money.

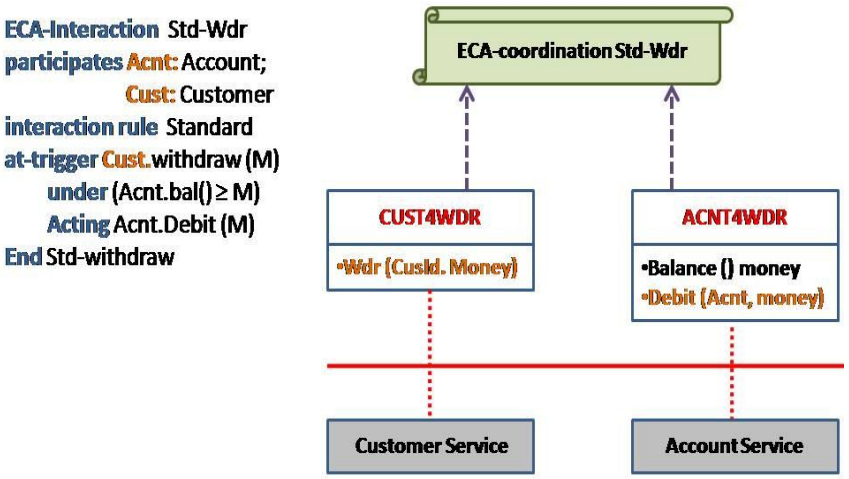


Figure 3.3: ECA-connector for standard withdrawal

Note that now no *condition* is to be assigned to the debit operation. That is, any conditions or constraints (i.e. business logic) are leveraged and externalised at the interaction level. As for the Std-Wdr interaction rule itself, as one customer and one account are needed for it to apply, it is necessary to assign respectively the instance *Cust* for the customer interface and *Acnt* for the account interface. To specify that the event originates with the customer, it is prefixed with the customer instance name, thus: *Cust.withdraw* (*M*). The informal condition that the balance should be greater than the requested amount is now formally specified as a first-order formula of the form: (*Acnt.bal* () ≥ *M*). Finally, the account debit operation is called when that condition holds and the event is triggered.

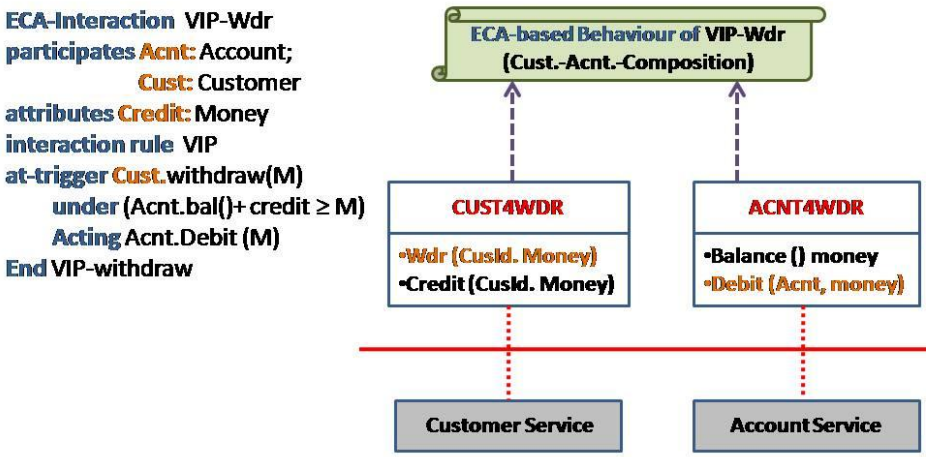


Figure 3.4: ECA-connector for VIP withdrawal

In the same spirit, as shown in Figure 3.4, there is a precise definition of the ECA-driven business rule associated with the credit withdrawal behaviour as an ECA-driven architectural service interaction. In contrast to the Std-Wdr customer-account interaction, the first requirement from the customer is a new attribute to reflect the presence of credit. The constraint is also now more flexible, as it involves the credit, so during a withdrawal the balance can go below zero. More specifically, the condition becomes a first-order formula of the form: (*Acnt.bal* () + *credit* ≥ *M*).

3.4 Towards Aspectual ECA-driven Architectural Interconnections

As stated above, the ECA-driven architectural interaction model allows for *design-time* adaptability, so that any ECA-driven interaction rule can be statically changed by another or new ones introduced. However, no mechanism has yet been proposed to cope with runtime adaptability. This section uses aspect-oriented concepts to demonstrate how to leverage this design-time adaptability towards runtime.

In contrast to the approaches that distinguish ordinary connectors from so-called aspectual ones (e.g. [14, 46]), the present work instead regards the incorporation of aspect-oriented concepts as a special mechanisation of any architectural conceptualisation. In other words, it is argued that any (ordinary) connector can handle cross-cutting concerns, when its operational interpretation is based on aspect-oriented mechanisms. This leads to a subsequent discussion of intuitive ideas on how the architectural conceptualisation should be endowed with aspect-oriented mechanisms to cope with dynamic adaptability.

First, given any event-driven ECA-centric architectural interaction, it is assumed at the instance level that specific partners participate in such interactions as service instances. Thus, an interaction instance corresponds to each such interacting service instance. In the banking example, customers *Cs1* and *Cs2* may have standard withdrawal interactions with accounts *Ac1* and *Ac2*, while *Cs3* and *Cs4* have VIP withdrawal interactions with accounts *Ac3* and *Ac4*.

The following are the main ideas relating to the runtime adaptability of such interaction instances. First, with respect to such interaction instances only, the proposal is to *intercept* and then externalise any triggering events as well as any required attribute and /or property

before going to the respective services, by bringing them first to their respective interfaces. Indeed, if such events were allowed to invoke directly the respective services, the related (composition) business logic which is already externalised in terms of ECA-driven rules could not be performed. Such instantiated service interfaces are then passed to the interaction level, where the appropriate ECA-driven rules will be performed. Finally, the resulting actions and states are to be dynamically and non-intrusively superposed onto the respective services.

These ideas are abstracted in Figure 3.5, which proposes a language-independent ECA-based conceptual architectural model with three levels: the service level, the interface level and the interaction level. Five successive steps are proposed in applying aspect-oriented mechanisms to this conceptualisation so that adaptability is supported.

(1) Intercept events and interface ingredients: In this step any events and information required from the services are intercepted at the interface level. The required information thus comprises the current values of the interface elements, applied to the intercepted instance. That is, intercepting the event *withdraw (Cs1, M)* requires the selection from the account interface of the current values of the *balance* and the *debit* operation.

(2) Propagate them to the interaction level: These intercepted interface ingredients are passed to the interaction level. That is, all elements required for the associated ECA-driven interaction will be ready after this step.

(3) Perform the selected ECA-driven service interaction: At the interaction level, the corresponding ECA-driven architectural interaction is now performed on the intercepted and agreed service partners.

(4) Dispatch the results to the service interfaces concerned: After the execution of the corresponding ECA-based rules, the resulting actions are collected and propagated to different service interfaces.

(5) Dynamically weave the ECA-driven interaction into service components: From the interfaces the interaction results are then woven into corresponding service components in a non-intrusive manner.

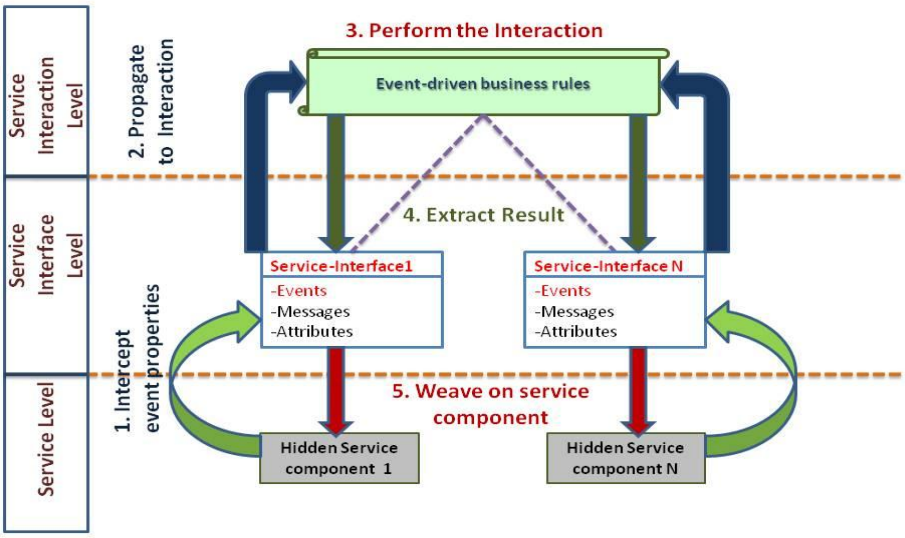


Figure 3.5: Stepwise abstract aspect orientation of ECA-based architectural model

In terms of aspect-oriented mechanisms, these logical steps can be interpreted as follows: (1) the ECA-driven interaction behaviours fulfil the role of (cross-cutting) advices; (2) the propagation process itself represents the point-cuts, that is, how to (intercept and) weave the advices; and (3) the join-points represent the basic behaviours at the component level, which are non-intrusively enriched with the ECA-driven interaction behaviours.

These steps summarise the abstract perception of applying aspect-oriented concepts to facilitate exogenous dynamic adaptability using ECA-driven architectural modelling for service-oriented business applications. The next chapter details how to automate this abstract perception practically and effectively using the Maude language and its reflection capabilities, as well as all required enhancements. It is important to note that this modelling of the aspect orientation of the approach is too abstract to be directly illustrated using the banking example, for instance.

3.5 Summary

This chapter has presented an integrated and progressive approach to modelling complex service-oriented applications, which have to be flexible, dynamically adaptable and knowledge-intensive. More specifically, starting from broad informal objectives and processes of any opportunistic cross-organisation alliance, it was first considered how different business activities involved in such a service-oriented and process-centric alliance should be informally governed by event-driven ECA business rules. To that end, an intuitive

and generic pattern of such ECA rules was proposed, where the participant services (as business entities), the triggering events, the constraints to be observed and the actions to be performed are to be declared. From this informal description of ECA-driven rules, the focus shifted to a closely service-oriented conceptualisation, where composition and interactions become the driving forces. More specifically, adapted architectural connectors were proposed as constituting a disciplined and interaction-centric behavioural conceptualisation of any business activity. In particular, it was shown how the above ECA-driven business rules could be converted to precise and still ECA-driven architectural connectors, with their roles detailed as explicit participating service interfaces in such a behavioural composition. Finally, to upgrade this design-time rule-centric adaptability of service composition towards runtime adaptability, the notion was introduced of a stepwise conceptual leveraging of such a service-oriented and rule-centric architectural conceptual model towards aspect orientation. This required the recapitulation of aspect-oriented concepts and mechanisms such as aspects, advices, point-cuts, join-points, weaving and unweaving.

To conclude this chapter, it should again be emphasised that the approach taken ranges over the descriptive and conceptual levels. That is to say, for the sake of the separation of concerns, there was no strong coupling of the approach with any specific formalisation at first stance. Consequently, different candidate formalisms can be considered to ensure the validation, certification and verification of the approach. The next chapter demonstrates and details a potential approach based on rewriting logic and the Maude language. Again, what makes the proposed approach generic and flexible is that other formalisms such as graph-transformation, Petri nets or temporal logic like ITL could also be applied. In any case, with the strengths of rewriting logic and Maude-based conceptualisation, it is relatively easy to shift to other formalisms. Finally, although the approach has so far been illustrated with a simplified banking example, the sixth chapter demonstrates how it allows the modelling of complex service-oriented applications such as e-commerce.

Chapter 4

Compliant Maude-based foundation and Validation of the Conceptual Modelling

4.1 Motivation

The previous chapter proposed an adapted conceptualisation for modelling dynamically adaptive, knowledge-intensive, service-oriented applications. More precisely, it first proposed ECA-driven business rules for governing the behavioural features of any business activity in a given service-oriented (cross-organisational) business process. Such event-driven ECA business rules were then smoothly leveraged towards a more disciplined yet ECA-compliant architectural modelling of interactions, where service interfaces and their inter-service interactions / compositions are addressed. Since such ECA-driven architectural interactions are completely decoupled from associated hidden services, adaptability is supported through the selection of the right ECA-driven architectural connector at any time while designing and composing services. Furthermore, in order also to support runtime adaptability directly in such an ECA-driven architectural model of interactions, the model was incrementally supported with aspect-oriented mechanisms. More precisely, as detailed in the final section of the previous chapter, five steps were proposed for enriching the model towards runtime adaptability. The resultant conceptual model of the dynamic, adaptive and rigorous development of service-oriented applications is (re)depicted in the grey-coloured box on the left-hand side of Figure 4.1.

Nevertheless, despite all the strengths and capabilities of this conceptual model of dynamically adaptive service-oriented applications, it remains limited to the descriptive level. That is, the model is not supported with operational mechanisms for its execution and cannot be directly served for validation and verification, as is desirable at the modelling level. In other words, alternatives can be adopted to make it executable and capable of supporting validation and verification. These alternatives can be summarised as being twofold. The first option is to map it directly onto the service technology, using Java or .Net technology, for instance [41]. A severe limitation of this technology-centric approach is the lack of any means to ensure its formal validation and verification. The second more logical

and ambitious alternative is to govern this descriptive conceptual model semantically with *a compliant formalisation*, which should itself be supported with inherent validation and verification capabilities. 'Compliant' here means that there must be no loss of the strengths and characteristics of the proposed conceptual model; indeed, they must be enriched with more features and capabilities.

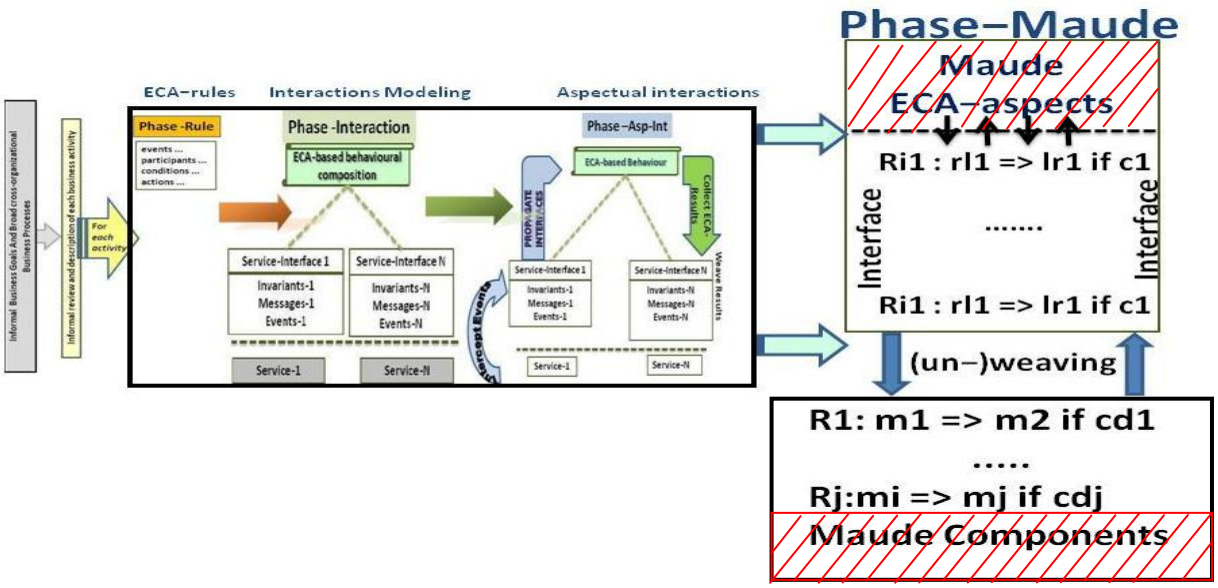


Figure 4.1: Stepwise abstract aspect orientation of Maude ECA-based architectural

The second promising alternative is pursued here in an attempt to govern the advanced ECA-driven conceptual model of aspectual architectural service interactions with a compliant formalisation that inherently supports validation and verification capabilities. More precisely, as graphically illustrated by the two boxes to the right-hand side of Figure 4.1, this chapter concerns the search for a compliant operational formalisation and validation. As indicated in the introductory chapter, the main tools adopted will be rewriting logic (RL) [71] and its reflective Maude language [27, 28]. Before advancing further in this chapter, it should be emphasised that other formalisms such as Petri nets, graph transformations or temporal-based settings could also be candidates for such formalisation.

Let us first recall some advantages of this formal framework over such other formal settings.

RL is a unified framework for true-concurrent systems, thus promoting the specification of distributed software-intensive systems. This is essential, since service-oriented applications are inherently distributed and concurrent.

Maude is highly efficient, allowing millions of rewritings per second for validation and rapid-prototyping purposes. This is also beneficial since service-oriented operations require huge computations (e.g. message and operation calls, invocation, reception and sending) while composing services.

With their intrinsic reflection capabilities [28], RL and Maude promote separation and explicit control of rule executions using strategies. As will be argued subsequently, these reflection capabilities represent a criterion for compliance with the aspect-oriented enrichment of the proposed conceptual model and thus the desired runtime adaptability. It should be added that to the best knowledge of the author, all existing formalisms (e.g. Petri nets, process algebras, graph transformations and temporal setting) lack this essential reflection capability and thus are not straightforwardly suitable to address dynamic adaptability, as is RL.

For certification purposes, RL has been endowed in recent years with a built-in LTL-based model checker in Maude [38], beside other temporal-based semantics such as the ones in [34]. That is to say, RL and Maude directly support verification by model checking, although this direction will not be further explored in this thesis, which will concentrate instead on validation.

Last but not least, Maude currently runs in both Linux and Windows environments, with a Java-based Windows-friendly workstation [67], which is the one predominantly used in the present implementation.

Nevertheless, despite all these capabilities and advantages over some other formalism, the Maude language as it is currently available does not directly supports the proposed approach in a compliant manner. That is, the conceptual approach cannot be directly interpreted in Maude without some loss of essence and strengths. In other words, the objective of this chapter is not a straightforward semantic mapping of the conceptual model onto the Maude language or a mere intuitive translation into Maude. More precisely, it aims to reflect more faithfully and formally the ECA-driven aspectual model of architectural interactions from the previous chapter, by first adapting the Maude language in several important respects so that it becomes compliant at that conceptual level.

In contrast to the explicit and *externalised* inter-service behavioural interactions provided by the proposed conceptual model, the Maude language does not at present scale up directly to

(service) componentisation. Indeed, as illustrated in the third chapter, Maude permits any software-intensive system to be specified and validated exclusively as a *holistic society* of concurrently running objects and messages. In other words, the Maude language must be extended so that it can inherently support service components and interfaces.

Since Maude does not scale up to a service component-oriented perspective, this also means that no explicit interactions between composed services are directly supported. That is to say, besides extending Maude to support service components and interfaces, this extension must be adapted so that it supports the explicit and externalised specification of ECA-driven interaction rules, as developed at the conceptual level.

Finally, as explained in chapter three, Maude is inherently endowed with reflection, which allows the programming and control of the performance of rules within any object-oriented system. However, when it comes to aspect-oriented mechanisms and dynamic adaptability, and how such reflection capabilities can be exploited in that direction, Maude is clearly completely inadequate. There is thus a need to further leverage the two extensions envisioned above towards supporting aspect-oriented mechanisms, by benefiting from the five descriptive steps proposed at the conceptual level.

This chapter thus details the attempt to modify Maude in order to overcome the above severe limitations and thus to formalise, validate and verify the conceptual model of ECA-driven aspectual architectural interactions, so that it can be used reliably to develop dynamic and adaptive service-oriented business applications. That is, capitalising on the flexibility of programming and extending Maude using Full Maude, the language must be equipped to address service componentisation and interfaces. Furthermore, the enriched Maude will have to be extended for service componentisation in order to externalise and validate ECA-driven behavioural interactions. Finally, a further extension of Maude concerns the intrinsic specification and reasoning about *runtime*-adaptive ECA-driven service interactions in a non-intrusive manner, that is, without disclosing service component internals, which are anyway by definition hidden from requestors and users.

More specifically, the next section of this chapter presents an extension to Maude so that service components and interfaces can be directly modelled, implemented and executed in a smooth manner. Secondly, on the basis of such Maude-based service components and interfaces, a method is proposed to deal with ECA-driven service interactions in an explicit

and transparent way. These two extensions will ensure a compliant, smooth and progressive mapping of the conceptual model of ECA-driven architectural interactions onto a Maude foundation, while preserving all the advantages of that conceptualisation. The third section focuses on the Maude-based mapping of the five steps proposed for integrating aspect-oriented mechanisms in the ECA-driven architectural conceptualisation for runtime adaptability. Maude's reflection capabilities will be employed for that purpose. More specifically, it will first be necessary to specify and implement the interception of triggering events for any externalised (ECA-driven) service interactions, from the associated service components. Secondly, on the basis of such intercepted events, a method is proposed to select the relevant service interfaces with their states from the running service components. Thirdly, there is an account of the selection and performance of the correct ECA-driven service interaction rules. The final step is to weave the resulting rule execution non-intrusively into the corresponding running service components via their interfaces.

As will be detailed subsequently, all these envisioned Maude extensions and mapping steps will be extensively explained and illustrated using the simplified banking example. It should also be emphasised that in order to enhance the clarity of the Maude programs concerned, the following simplifications will be adopted in this chapter and the next. Proposed Maude formalisations will not be displayed in full as screen-shot program code from the Maude environment. Instead, only those lines that are essential to convey the essential aspects of any formalisation and programming of (the extended) Maude will be extracted and presented. Line numbers will also be added to facilitate the discussion of individual parts of the selected Maude code. For sake of completeness, appendixes A and B reproduce the complete original code which was run and validated on concrete inputs using the Maude environment and its workstation.

4.2 Extending Maude to Service Components and Interfaces

Recall that Maude supports only a holistic object community configuration, where objects are represented as *indivisible* and monolithic tuples of the form

$$\langle \mathrm{Id} : C \mid \mathrm{at}_1 : v_1 , \cdots , \mathrm{at}_k : \mathrm{v}_k \rangle$$

In other words, the Maude configuration is far from being ready to cope with service componentisations and their explicit interfaces, which this project aims to do. To overcome this serious limitation, this section is devoted to the service componentisation leveraging of

the Maude language. That is, in place of the holistic object-oriented configuration of Maude presented above; the proposal is for a more advanced service-component-based configuration with the following features.

Scoped service component properties: To allow extraction of observed interfaces, it is proposed to replace the object state by a service component state of the form:

$$\langle \text{Id: } C \mid \ atl_1 \ : vl_1 \ , .., \ atl_k \ : vl_k \ , \ atbs_1 \ : vs_1 \ , .., \ atbs_i \ : vs_i \ \rangle,$$

where atl_i are assumed to be local properties, thus hidden from the outside, whereas $atbs_j$ are the observed properties of that service component state.

Service component state splitting / recombining: The extraction of any observed properties from local ones is allowed, so that observed interfaces can be built on the fly. For that purpose, each service component state is endowed with an inference rule that permits it to be split or recombined at need. This splitting / recombination can be represented by the following (inference) axiom:

$$\langle \text{Id} \mid prs_1 \ , prs_2 \ \rangle = \langle \text{Id} \mid prs_1 \ \rangle \ \langle \text{Id} \mid prs_2 \ \rangle$$

where prs_i is the abbreviation of different pairs of 'attribute: value'.

Imported / exported messages and events: Besides the above ability of observed state properties, messages are also allowed to be observed. There is thus a distinction between local messages (to that service component) and observed ones, which have to participate in external interactions. Moreover, events are distinguished as triggering messages, that is, messages appearing only on the left-hand side of a given rule.

4.2.1 Service Components in Maude

Concretely, instead of the usual holistic object configuration, this work introduces a new Maude configuration named *service component configuration*, whose complete specification is depicted in Figure A.1 in appendix A, with the essential part shown below.

```
1. mod CMP_GNR is
2. sorts StatCMP obs_StatCMP loc_StatCMP
   ConfCMP obs_ConfCMP loc_ConfCMP.
3. subsort obs_StatCMP loc_StatCMP< StatCMP.
17. op_ _: ConfCMP ConfCMP → ConfCMP
                [ctor assoc comm].
. . .
21. rl [SplitAT] : < I | prs1 , prs2 >
                ⇒ < I | prs1 > < I | prs2 >.
```

```
 . . .                                                  22.rl [RecombineAT]:< I | prs1> <I | prs2>
15. op <_ | _ > : CMPid loc_Prop → loc_StatCMP                  ⇒ < I | prs1, prs2 >.
16. op <_ | _> :CMPid obs_Prop →obs_StatCMP      . . .
```

First, as given in lines 2 and 3, a distinction is made between sorts obs_StatCMP and loc_StatCMP to separate the observed and local properties in any given service component state. This distinction is then exploited to specify how to construct the observed part of any service component state (line 16), the local state part (resp. line 15) and the merging of the two parts (line 17). An equivalent distinction is made between imported / exported observed and local messages, but this part is not shown here. ConfCMP is further defined to refer to service components, instead of the usual Maude (object) configuration discussed in chapter three. Similarly, instead of objects, StatCMP is defined as referring to service component states.

Second, two corresponding rules are proposed to specify state splitting / recombination. The split rule SplitAT, given in line 21, takes a merged state and allows it to be split into two parts. Conversely, the recombining rule RecombineAT, defined in line 22, takes two split parts and allows their recombination. It is important to point out that these two rules are to be deployed only through an adequate (meta-)strategy, which will be detailed in the third section. Otherwise, a lack of control might easily result in looping, where recombined elements would be split again. As already mentioned, the complete specification of this generic service component pattern, with all of the sorts, operations and equations performed in the Maude Workstation environment, is depicted as a screen-shot in Appendix A, Figure A.1.

To recapitulate, from now on this Maude-based service component configuration will be used instead of the unsuitable standard holistic object-oriented configuration. It is important to recall that the service component state is invoked because the aim is to externalise the business logic (in terms of interacting event-driven business rules) and therefore the service components are not completely hidden. There should at least be explicit access to the following features: (1) all the observed properties, which correspond to observed attributes, messages and events; (2) the basic behavioural rules that reflect the intended state changes through the execution of messages. As will now be demonstrated, only observed features will be addressed in service interfaces.

4.2.2 Interfaces in Maude

Using the Maude-based service component configuration, it becomes straightforward to define service interfaces. First, to keep track of the service component running instances, the respective required service interface state takes the following form.

[IntfName | Interface-conf.]

The service interface name IntfName is thus a specific instantiation of the (invoked / selected) service component. Interface-conf is to be composed exclusively of observed features, that is, required observed properties, messages and events to be used in the appropriate ECA-driven interaction rule. This specific service interface structure is implemented in the partial code reproduced below (lines 11-12), extracted from the complete Maude-based service interface specification given in Appendix A, Figure A.2. Note that the service component specification is required here and is incorporated using the include (inc) Maude command.

```
1. mod INTF_GNR is
2. inc CMP_GNR . ···
11. op [ _ | _ ] : Intf_NM ConfINTF → EX_ConfIntf.
12. op _ _ : ConfINTF ConfINTF → ConfINTF [ctor config assoc comm].
13. op subsume( _,_ ) : ConfINTF Intf_NM → EX_ConfIntf.
14. op belong( _,_ ) : Iid IidL → Bool .
15. op weaveCfIntf( _,_ ) : EX_ConfIntf ConfCMP -> ConfCMP.
16. op intercept(_,_ ) : ConfINTF ConfCMP → ConfCMP . ···
22. rl [belong] : belong(I1,I2 ◦ IL) ⇒ (if I1 == I2 then true
23.               else if IL == nil then false else belong(I1, IL) fi fi).
24. rl [Subsume] : subsume(Cfintf, INM) ⇒ [INM | Cfintf] .
25. rl [weaveCfIntf ] : weaveCfIntf([INM | Cfintf], Cfcmp) => Cfintf Cfcmp.
26. rl [intercept] : intercept(Cfintf, Cfintf Cfcmp) ⇒ Cfcmp .
27. endm
```

First, in line 11 the above service interface structure is defined, where ConfINTF represents the running service interface state selected from the service component as fitting the properties required for the interaction. As depicted in line 24, the subsume extra operation allows a service interface instance name to be associated with any selected service component configuration as required.

Besides the specification of such a service interface structure in this extended Maude, consideration must be given to how to intercept events and other properties of any involved service component and divert them towards service interfaces to facilitate the explicit service

interaction. Using the service component extension to Maude, the aim of such interception is to formalise the first of the five steps detailed in the previous chapter towards aspect-oriented mechanisation. In this sense, the interception rule intercept in line 26 extracts from the service component state any part that concerns the interaction, such as events and properties required for any specific ECA-driven interaction rule (to be formalised later). Through the rule Subsume in line 24, the supposedly intercepted events and/or properties are transformed into the special form of service interface states, so that they can be directly used at the interaction level. The weaving rule weaveCfIntf proceeds in the other sense, that is, it allows the service component state to be enriched with the resulting service interface state, after the interaction has been executed as demonstrated later. Finally, the belong rule given in line 22 and 23 is important, as it permits the dynamic checking of which interface instances are under which interactions. Consequently, it permits the intelligent steering of the interception in such a way that only partner instances involved in the interaction are intercepted from among the service components concerned.

4.2.3 Application to the Banking Example

To illustrate this enriched configuration of Maude-based service components and interfaces, the banking case study developed in the previous chapter is used. As an illustration, this subsection reports on the essential features of the account service component, while noting that it's complete specification and the customer service one have been performed in the Maude Workstation environment and are given in Appendix A.

In Appendix A, Figure A.5 concerning the account service, as given in lines 10-11, this illustration considers two properties of the accounts: the balance bal as an observed one and the limit limt as a local one. Further, as given in lines 12-15, it considers credit, debit and transfer as observed messages, whereas change-of-limit is a local (hidden) message. It is important to emphasise that the debit rule now contains *no conditions*. Indeed, as detailed in the previous chapter, the objective of this work consists in externalising any business logic and thus any interacting business rule at the interaction level. Therefore, the conditions will be expressed as ECA-driven service interaction rules, and then dynamically enforced using an aspect-oriented mechanism. The debit rule at the service component level now states only that the balance is to be decremented by the requested amount. The same reasoning applies to the transfer rule, which now contains no constraint or logic assumed to be externalised for adaptability purposes.

```
1. mod ACNT CMP is  ···
10.op bal: _: Int → obs_ Prop .
11.op limt:_ : Int → loc_Prop .
12.op Crd( _,_ ) : AcntId Int → CRDT .
13.op Db( _,_ ) : AcntId Int → DBT .
14.op ChgL( _,_ ) : AcntId Int → CHGL .
15.op Trs( _,_,_ ) : AcntId AcntId Int →TRS.
···
22.rl [credit] : Crd( A, M ) < A | bal: B >
                 ⇒ < A | bal: B + M >.
23.rl [debit] : Db(A, M) < A | bal: B >
                 ⇒ < A | bal: B - M >.
24.rl [chgl] : ChgL( A, L1 ) < A | limt: L >
                 ⇒ < A | limt: L1 >.
25.rl [transfer] : Trs(A, A1, M) < A | bal: B >
  < A1 | bal: B1 > ⇒ < A | bal: B - M > < A1 | bal:
B1 + M >  ···
```

The associated service account interface to be involved in the ECA-driven interaction rule for (standard / VIP) withdrawal is then formalised. As depicted below, the corresponding part of that Maude-based service account interface *ACNT_INTF_GNR*, extracted from the complete specification in Appendix A, Figure A.7, can be highlighted as follows. The debit and balance properties are first required, but the balance has to be intercepted only for those instances that are in (standard or VIP) agreements at the interaction level. This interception is implemented through the rules getCfIntfbal and getCfIntfbalf in lines 18-22.

```
1. mod ACNT_ INTF_GNR is  ···
8. op ACNT : → Intf_NM .
9. op Db( _,_ ) : AcntId Int → DBT [ctor].
10. op bal:_ : Int → obs_Prop [ctor gather (&)] .  ···
18. rl [getCfIntfbal] : getCfIntfbal(< AC | bal: B > Cfcp, AcntsL)
        ⇒ (if belong(AC, AcntsL)  then < AC | bal: B > getCfIntfbal(Cfcp, AcntsL)
         else getCfIntfbal(Cfcp, AcntsL) fi) .
22. rl [getCfIntfbalf ] : getCfIntfbalf(Cfcpf getCfIntfbal(Cfcp, AcntsL)) ⇒ Cfcpf .
23. endm
```

4.3 Leveraging Maude to ECA-driven Behavioural Service Interactions

The previous sections have proposed a newly adapted Maude configuration that supports the formalisation of service components and interfaces. Furthermore, the service interface formalisation was endowed with anticipated rules for intercepting and weaving required properties and messages / events from service components. It is now possible to formalise and validate the descriptive conceptual model of ECA-driven service interactions that was put forward at the conceptual level in the previous chapter.

To that end, the first proposal is for an adapted Maude-based structural representation that captures all elements from such ECA-driven interaction rules. It will be recalled that the elements composing any ECA-driven interaction rule are these:

(1) The name of the ECA-driven rule itself (e.g. *WdrStd* or *WdrViP* for the banking example). As explained at the conceptual level, each such name should be uniquely used to characterise any specific ECA-driven rule and the business activity to which it is assigned.

(2) The identifiers of the participating service interfaces (e.g. *ACNT* and *CUST* in this case). These are essential, as they reflect the contract character of such rules and allow them subsequently to be instantiated as specific instances (e.g. *ac1* and *cs1* under *WdrStd* and *ac2* and *cs3* under *WdrViP*).

(3) Any specific information such as properties and operations which may be required besides those from the service interfaces involved (e.g. the credit attribute). For instance, if the credit is to be manipulated outside the customer, it must be declared at the interaction level, so that it can be updated and adapted as required.

(4) The required properties, messages and events for each service interface involved.

In order to gather flexibly all these elements comprising an ECA-driven service interaction rule, the following Maude-based adapted structure is proposed, where the resulting ECA-driven structure is an algebraic tuple of the form:

$$[InterName \,||\, (partner\_ids(\$partner\_ids)^{*})@inter\_infos]\ (\&[Partner_i\,|partner\_infos])+$$

This begins with the interaction name, followed by the participating service interface identifiers, separated, arbitrarily, by the symbol operator $. It is then necessary to specify any proper information required at the interaction level, such as extra attributes, constants and operations. This information is prefixed by the symbol operator @. Finally, any information needed from each service interface, as partner in this interaction, is specified.

This Maude-based structure is first based on the service component and interface configuration. It is recursively defined through the following (abbreviated) Maude module, named *ASP_COORD*. Moreover, to exhibit a maximum of concurrency, different parts of that ECA-oriented algebraic term are allowed to be split and recombined as required. These splitting and recombination capabilities are captured using respective rules given in that specification, namely Split_CfIntf defined in line 25 and Recombin_CfIntf in line 26.

Furthermore, to prepare superposition of the resulting interaction on different service components, the extraction of any part (related to a specific service interface) from that

service interaction or coordination configuration is permitted, using the rule extractCFIntf given in line 27. The complete specification of that generic service coordination configuration, as implemented, performed and validated in the Maude Workstation environment is depicted in Appendix A, Figure A.3.

```
1. mod ASP_COORD is
2. inc INTF_GNR
. . .
15. op _$_ : PartnerIds PartnerIds → PartnerIds [assoc comm] .
16. op _;_ : Attributes Attributes → Attributes [assoc comm] .
17. op_. _ : CoorOperations CoorOperations → CoorOperations [assoc comm] .
18. op _@ _: PartnerIds Attributes → PartnerAttrs .
19. op [_ ||_ ] : Coord_NM PartnerAttrs → ObjCoord .
20. op extractCfIntf(_ , _ ) : ConfCoord Intf NM → EX_ConfIntf .
21. op _&_ : ConfCoord ConfCoord → ConfCoord [assoc comm] . . . .
25. rl [Split_CfIntf ] : [Inm | CfI1 CfI2] ⇒ [Inm | CfI1] & [Inm | CfI2] .
26. rl [Recombin_CfIntf ] : [Inm | CfI1] & [Inm | CfI2] ⇒ [Inm | CfI1 CfI2] .
27. rl [extractCfIntf ] : extractCfIntf([Inm | CfI1] & CfCt, Inm) ⇒ [Inm | CfI1] .
28. endm
```

Example: Capitalising on that generic formalisation of ECA-driven service interactions, the next step is to apply the ECA-driven service interaction rules associated with the withdrawal business activity, namely the (standard and) VIP rule elaborated in the previous chapter. The corresponding complete formalisation, implementation and validation are given in Appendix A, Figure A.8. To report the essentials of this formalisation, the part related to the VIP-withdrawal interaction has been extracted and is reproduced below.

First, a corresponding interaction name is chosen: *WdrVip* (line 6). The interaction specific attribute *Credit* (*crd*) in line 7 is also defined for use in *WrdVip*. As a generic service partner instance identity, *CS* has been chosen for the customer service and *AC* for the account service. The left-hand side of the VIP withdrawal rule (from lines 11 to 14) says that to apply that rule it is above all necessary to intercept any withdrawal event sent to the customer partner *CS*. This interception[1], as already discussed, is carried out at the customer interface *CUST* level. Since the *balance* of the service account is also required, it has to be shifted up from the account service component. The right-hand side of the VIP withdrawal rule first imposes the required constraint, that is, the credit *crd* plus balance must be greater than the requested amount to be withdrawn. The application of the rule in that case allows the

[1] That is, CS and AC are part of that VIP-withdrawal contract.

consuming of the withdrawal event from the customer interface and the sending or production of the debit message.

That is to say, the withdrawal event is consumed, leaving an empty customer interface, namely [*CUST* | *nil*]). On the other hand, for the account service interface, the balance is kept unchanged, whereas a new debit message is to be sent to that account, namely [*ACNT* | < *AC*|*bal : B* > *Db(AC, M)*].

```
1. mod COORD_WdrVip is
. . .
6.  op WdrVip : → Coord_NM .
7.  op crd:_ : Int → Attribute [ctor gather (&)] .
. . .
11.  crl [WdrVip] : [WdrVip || (CS $ AC) @ crd: C] & [ CUST | Wdr(CS, M) ]
12.     & [ ACNT | < AC | bal: B > ]
          ⇒ [WdrVip || (CS $ AC) @ crd: C] & [ ACNT | < AC | bal: B > Db(AC, M)]
14.    & [CUST | nil] if (B + C) ≥ M .
15.  endm
```

4.4 Runtime Adaptability of Aspectual ECA-driven Service Interactions

So far it has been shown how service components, interfaces and ECA-driven service interactions can be formally specified and executed, by extending the Maude language accordingly. The last step towards the formalisation of the desired non-intrusive dynamic adaptability of the ECA-driven service interactions consists in judiciously composing all of these new and advanced Maude ingredients (e.g. service components, interfaces and ECA-driven interaction rules). In the following, first, an informal explanation of how runtime adaptability is achieved with our aspectual Maude is presented. Then, this stepwise operational conceptualisation with the running banking example is illustrated.

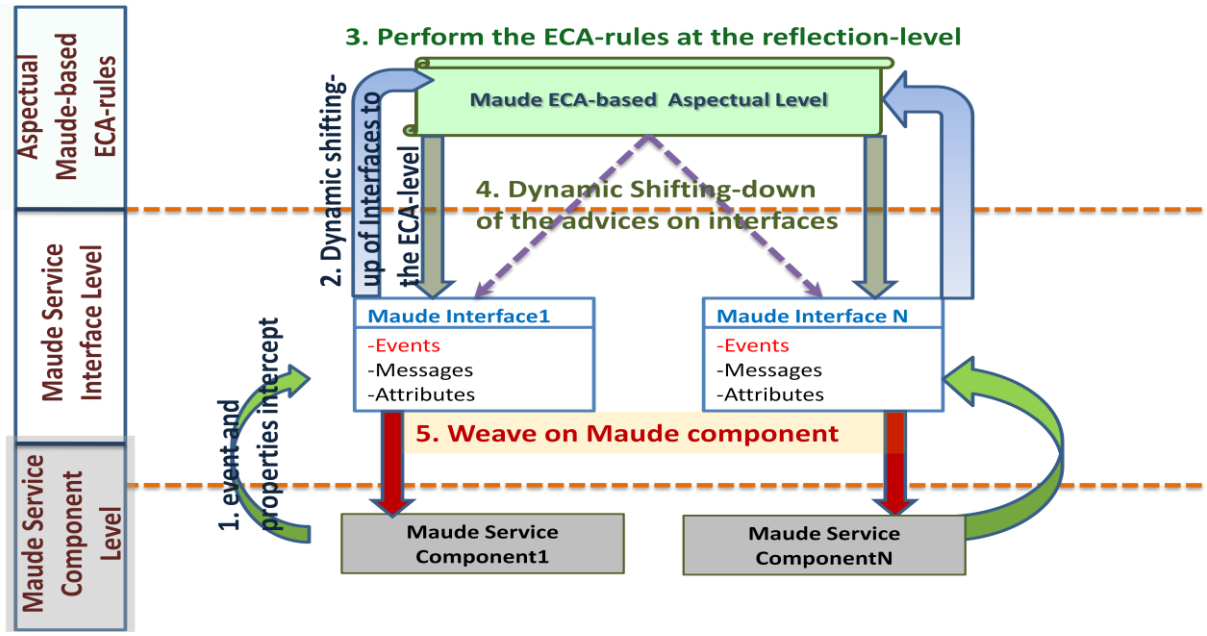


Figure 4.2. Dynamic adaptability of services using Aspectual Maude

4.4.1 Dynamic Adaptability of Services with Aspectual Maude: Informal Presentation

Figure 4.2 illustrate the concrete steps for the deployment of the abstract aspect-oriented stepwise architecture for dynamic adaptability which conceived in Figure 4.1. This deployment or realisation is based on the forwarded aspectual Maude. In the following these steps will be explained in some details:

First some service components assume have to be given, implemented using the extended Maude, as developed in the previous subsections. That is, service components structure is conceived through different operations and messages, whereas their internal behaviour is specified using appropriate rewrite rules. The service properties are defined as observed attributes. From the outside, only the messages, the events and the observed properties belong to the service component interface. All other elements, such as the rules and local properties are hidden.

Then service component assume that have been given is invoked through its observed events and triggering messages. In contrast to the usual SOA request-response exchanges of (sent-receive) messages, such triggering events and messages are first *intercepted* from the corresponding discovered service components. They are then passed or propagated to the aspectual-level, where ECA-driven service components are to be invoked.

Depending on the requested service instances and their contract at the aspectual-level, the concerned rules are selected. For executing such selected rules, the required interfaces from different involved service components are then propagated to the aspectual-level

Once the correct rules are performed at the aspectual-level, the result are then dispatched on the associated service interfaces.

Finally, these extracted results are then woven in a non-intrusive manner on the corresponding Maude-based service components, where the right hidden rules are to execute.

4.4.2 Dynamic Adaptability of Services with Aspectual Maude: Detailed Steps Execution

For that purpose, use is made of the reflection capability of the Maude environment, as discussed in chapter three. More precisely, in order to be able to intercept events non-intrusively, to execute them and then weave the resulting ECA-driven actions into respective service components, the five abstract aspect-oriented steps developed in the previous chapter should be followed. More specifically, after a deep exploration of the Maude reflection capabilities, it was concluded that the following directives should be respected to reflect faithfully these proposed aspect-oriented steps:

Any participating service component states (i.e. configurations) should first be split to prepare the interception of any required events and observed properties and messages. In this sense, all local properties should be hidden and thus not concerned by the observed interfaces. Further, splitting all elements in a given service component state will promote a maximum of concurrency while performing the next steps.

It is then necessary to intercept from these service component states all events and required information related to the ECA-driven service interaction rules of interest. That is, depending on the specific or instantiated service partners agreeing to interact through such rules, the right events and other required information have to be extracted from the service interfaces.

This interception is to be followed by the propagation of these intercepted service interface states to the interaction level. This step is to be ensured by the extra rules proposed for the service interfaces (e.g. [subsume], [intercept] and [weaveCfIntf]).

The relevant ECA-driven service interaction rules must then be applied to these interface states and participating instances, which may be enriched with specific information at the interaction level.

Finally, the results of performing the interaction rule on the respective service interfaces should be dispatched and then woven into the respective running service component states.

As mentioned already, in terms of aspect-oriented mechanisms, ECA-driven interaction behaviours play the role of (cross-cutting) advices. The reflection strategy itself represents the point-cuts, that is, how to (intercept and) weave the advices, while the join-points represent the rules at the service component level, which are non-intrusively enriched with those externalised ECA-driven interaction rules. For instance, the debit method is externally enriched in this case with the balance sufficiency (plus the credit for the VIP). The following simple banking example illustrates the general pattern of the above reflection strategy.

Example: The above running banking example thus requires certain main steps to be followed in writing a strategy for intercepting events and required interface information from service components and propagating them to the interaction level, then performing and weaving the results into the participating service components. The complete Maude strategy specification for the ECA-driven (standard and VIP) withdrawal interactions is specified in Appendix A, Figure A.9. Below is an extract showing the essential part to emphasise the strategy which non-intrusively performs the ECA-driven VIP withdrawal interaction rule.

```
1. mod ASP_WDR_Str is
. . .
11. op Compute : Term Nat → Term .
12. ceq Compute(T, N)
13.    = if(N == 1) then
14.    (if(SplitAT? :: Result4Tuple)
15. then Compute(getTerm(SplitAT?), N)
16.  else if(belong? :: Result4Tuple)
17. then Compute(getTerm(belong?),N)
18. else if(getCfIntfwdr?::Result4Tuple)
19. then Compute(getTerm(getCfIntfwdr?),N)
20. else if(getCfIntfwdrf?::Result4Tuple)
21. then Compute(getTerm(getCfIntfwdrf?),N)
22. else if(getCfIntfbal?::Result4Tuple)
23. then Compute(getCfIntfbal?),N)
24. else if(getCfIntfbalf?::Result4Tuple)
25. then Compute(getTerm(getCfIntfbalf?),N)
30. else if(Split_CfIntf? :: Result4Tuple)
31.  then Compute(getTerm(Split_CfIntf?),N)
32. else if(WdrVip? :: Result4Tuple)
33. then Compute(getTerm(WdrVip?), N)
. . .
35. else  (if (Recombin_CfIntf? :: Result4Tuple)
36. then Compute(getTerm(Recombin_CfIntf?),0)
37. else if extractCfIntf? :: Result4Tuple
38. then Compute(getTerm(extractCfIntf?), 0)
39.else if (weaveCfIntf? :: Result4Tuple)
40.then (if(debit? :: Result4Tuple)
41. then Compute(getTerm(debit?),0)
42.else Compute(getTerm(weaveCfIntf?),0)fi)
43. else if(RemoveNil? :: Result4Tuple)
44. then Compute(getTerm(RemoveNil?),0)
45. else if(RecombineAT? :: Result4Tuple)
```

```
26. else if(intercept? :: Result4Tuple)          46. then Compute(getTerm(RecombineAT?),0)
27. then Compute(getTerm(intercept?),N)
28. else if(Subsume? :: Result4Tuple)
29.then Compute(getTerm(Subsume?), N)
```

This strategy is in accordance with the general guidelines given above and performs the following concrete steps. First the SplitAT rule (lines 14 and 15) is applied to all participating service component configurations, namely the Account and Customer services. The belong rule (lines 16 and 17) is then used to extract only the state parts which are in agreements by using the agreed-on list of service entity instances (i.e. concrete accounts and customers, as illustrated below). Thirdly, the rules getCfIntfwdr and getCfIntfwdrf (from line 18 to line 21) are used to intercept all withdraw events from the customer service component states in agreement via VIP. In the same manner, the rules getCfIntfbal and getCfIntfbalf (from lines 22 to 25) are used to extract all relevant balances from the account service component state which are in VIP agreement with their customer. Next, these intercepted events and/or properties are removed from the service component instances, through the intercept rule in lines 26-27. These account and customer service interface states are first prepared to the specific structure of the ECA-driven interaction rules, using the rule Subsume (lines 28-29). They are then merged into the ECA-driven interaction state, after being split through the rule Split_CfIntf (lines 30-31) to exhibit a maximum of concurrency. At this stage the interaction rule is to be performed, that is, the WdrVip (lines 32-33) is performed. After this rule has been executed at the interaction level, any split parts of different resulting interfaces must be recombined, using the rule Recombin_CfIntf (lines 35-36). Different service interface states are then separated using the rule extractCfIntf (lines 37-38). Finally, the resulting Customer and Account actions are woven into the respective service components, namely the account and the customer services, using the rule weaveCfIntf as given from lines 39 to 42. The rules associated with such different actions are then performed at the service component level. In this case, the debit rule is to be performed, but only after it has been ensured that its VIP business logic holds at the interaction level. Finally, any empty elements must be cleaned and all split elements recombined through the RemoveNil (lines 43-44) and RecombineAT (lines 45-46) rules.

4.5 Summary

The following important issues have been addressed in this chapter. First, a foundation was adopted, based on rewriting logic and its reflective language, Maude, to govern the proposed conceptual approach to runtime adaptive and knowledge-intensive service-oriented systems development. The full potential of Maude was exploited to this end; indeed, it was enhanced and extended to make it compliant with the conceptual approach. That is, its specification was leveraged with new mechanisms to reflect the main service concepts, namely service components and service interfaces. The notion of ECA-driven service interactions was then captured in an explicit and externalised manner. Finally, Maude's reflection capabilities were exploited to propose a stepwise formalisation to cope with the dynamic adaptability of governing rule-based interactions, while interacting service components are running on specific configurations.

The formalisation and leveraging of Maude to fit the conceptual approach were illustrated with the simplified banking example developed at the conceptual level in the previous chapter. All Maude formalisations were validated with concrete component, interface and rule configurations and instances. That is, although only the essential Maude formalisation of each concept was presented, the complete codes and their validation results are available in the appendixes. Thus, interested readers (including Maude specialists) will find all the resources required to experience for themselves the entire approach, its formalisation and its validation.

The next chapter takes a step further towards the practical application of the approach. More precisely, it applies all stages of the approach, including those presented in this chapter (e.g. Maude-based formalisation and validation), to a typical service-oriented e-commerce application.

Chapter 5

Validating the Approach with an E-commerce Case Study

This chapter aims to demonstrate the practicability of the approach using a non-trivial case study other than the simplified banking example which was used for proof of concepts throughout the previous chapters. More precisely, it presents in detail a medium-size case study from the domain of e-commerce, dealing with online service-oriented shopping for goods. More precisely, the objectives of undertaking this application of the proposed approach are fourfold:

The first is to validate the scalability and the practicability of the stepwise approach outlined in the two previous chapters. That is, without such a case study, it would be hard to emphasise all the practical strengths and capabilities of the proposed approach and the possible limitations. Indeed, while all the steps of the approach have been illustrated using the simplified banking withdrawal example, it could not be used to demonstrate how several business activities can be put together to compose a service-oriented business process such as an e-commerce application. Nor was it possible to show how to interweave non-trivial business activities flexibly into a business process. Besides, this chapter will address the case where more than two service interfaces are to participate in a given business activity. All these issues and challenges will be tackled through the e-commerce case study.

A case study of the application of the proposed approach to an e-commerce domain has been chosen to leverage its practicability, at least in respect of the following important features. First, widely known e-commerce applications such C2B and particularly B2B have provided a major motivation for the emergence of service-oriented technology. Second, e-commerce applications must be flexible and competitive, so that cross-organisations offering such services stay successful and competitive while flourishing. Third, as will be detailed in this chapter, e-commerce applications are knowledge-intensive, their business logic and behaviour being mainly governed by a very large and rapidly changing number of ECA-driven business rules. Last but not least, runtime adaptability is more than an option due to the long-running and transactional nature of such service-oriented e-commerce applications

and the rapidly changing nature of customer and provider profiles, as well as application requirements.

A further aim of applying the approach to at least two different domains, namely the financial (simplified banking) application and this e-commerce application, is to demonstrate its generality and adequacy for most complex and agile service-oriented applications. It should be noted here that the long-term objective is to include applications such as e-health and e-government and to propose specific patterns and guidelines for each of these service-oriented application domains.

As will be detailed in the remaining sections of this chapter, this case study is considered typical in a number of ways. First, it is typically service-oriented, as it concerns online e-shopping. Second, the application itself is composed of more than eight non-trivial business activities (e.g. product request, product offer, request confirmation, request cancellation, goods shipment, and customer profile update). Third, each of these activities is governed by several ECA-driven business rules, twelve of which are considered here, although this number could easily be augmented. Fourth, to express the ECA-driven behavioural interactions governing such rules, more than ten service components come into play, with complex service interfaces (and significant internal service computations) being required.

Having set out the motivations and objectives of this chapter, the remaining sections are organised as follows. First, there is an informal description of the service-oriented application being considered: a variant of e-commerce shopping for goods. Moreover, since an activity-based approach is taken, there is an informal description of each of the component business activities as well as the whole business process. After this informal description of different business activities, the approach is applied by following the steps as detailed in Figure 4.1 and recalled here as Figure 5.1. That is to say, a complete section is devoted to each business activity, comprising at least four subsections, namely: (1) the informal description of the selected ECA-driven business rules governing that business activity; (2) the smooth shifting of these informal rules to the service-oriented architectural interaction level; (3) the formalisation of the service-oriented ECA-driven interaction model in the extended service component Maude; and (4) the dynamic interception of these service-oriented interaction rules and their weaving into running service components. As six business activities are detailed, six sections follow the informal description of the case study. It should finally be noted that in the same spirit as in the previous chapter, not all Maude

formalisations are depicted; instead, the essential elements are extracted and discussed. However, with respect to each business activity, the complete formalisation, implementation, execution and validation are given in Appendix B.

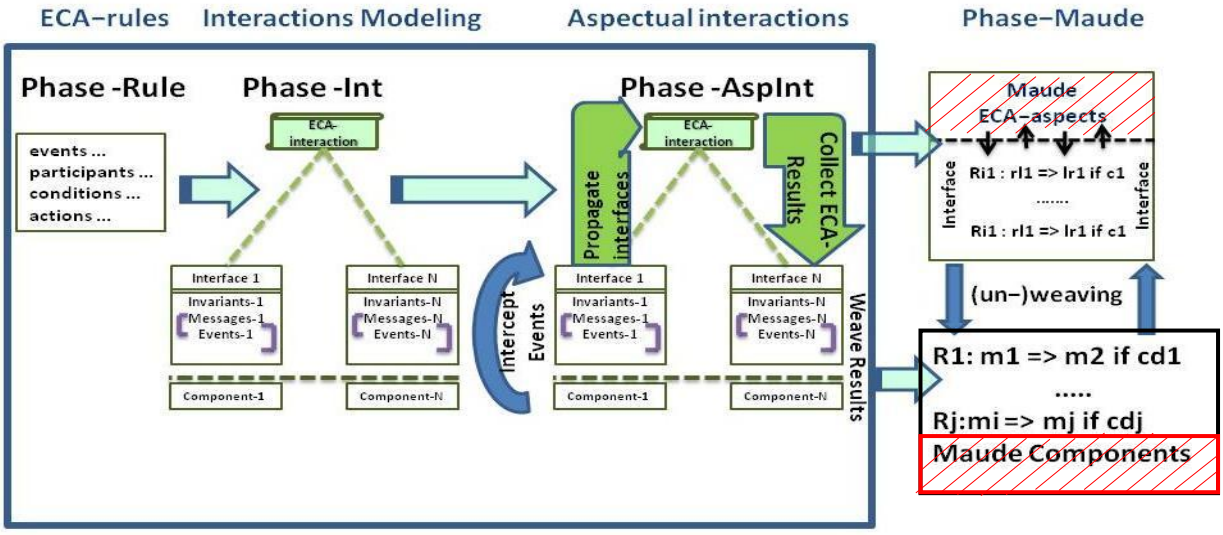


Figure 5.1: Recapitulation of the phases of the proposed approach

5.1 E-commerce Application: Informal Presentation

This chapter considers the following online shopping case study from the area of service-oriented e-commerce. As depicted in Figure 5.2, the business activities composing this business application are as follows:

Order product: This activity is mainly triggered by events such as *order-products* from the customer. In thus consists in putting the customer or the requester in direct interaction with the provider(s) of such products. It is important to mention in this initial description of the business activity that no specific or concrete provider is identified; the behavioural ECA-driven requirements of such providers are detailed during the next phase. Following the present approach, at no stage is there any reference to a concrete provider of specific Web services, since it operates at the modelling level.

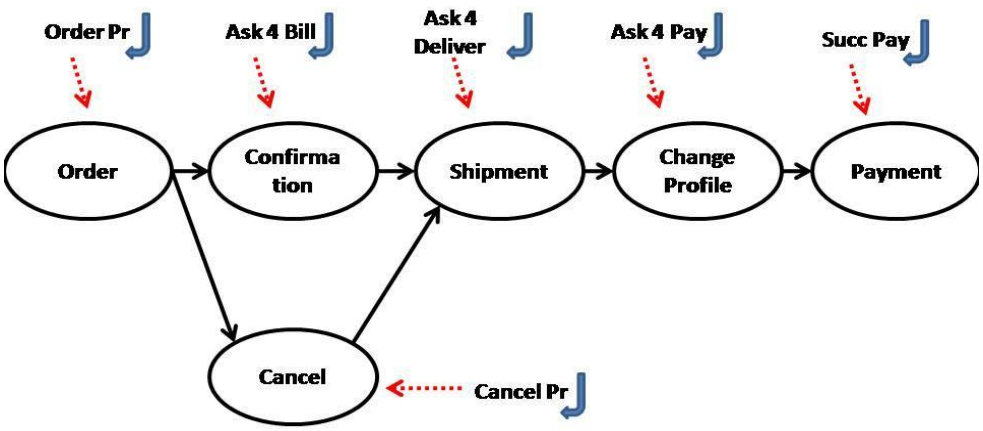


Figure 5.2: The business activities of the e-commerce case study and their ordering

Confirmation: Upon requesting a product or goods, the customer should receive an offer from the provider. Then, the confirmation activity is to be performed, putting the requester and the provider into interaction, where specific agreement on the prices, possible discounts and other constraints are to be negotiated and clarified.

Cancellation: At any stage of this e-shopping process, the customer can proceed to cancel some or all of the requested goods. However, the penalty and the amount refunded will depend on the stage at which the cancellation occurs. For instance, after payment has been made, a refund is usually difficult or impossible, whereas immediately after accepting an offer, only a specific percentage of the global price or penalty is to be paid.

Shipment: This business activity may be optional for customers located close to the premises of the provider or its representative. It depends also on the form of shipments (express, normal, airfreight etc). In other words, rules governing this activity range from simple to complex negotiated ones.

Change profile: Although this activity is usually expressed implicitly, it was decided to separate and address this business activity in a transparent and rule-based manner, since the aim is the explicit externalisation of any business logic. In other words, the customer him/ herself should participate with the provider in expressing the rules governing any profile change, for instance from normal to silver to gold.

Payment: The last business activity concerns the payment of the bill, where different rules governing the discounts and other (dis)advantages could be negotiated and formulated with the provider. It is worth noting here that different forms of payment are possible, such as bank or credit cards, transfer and cash.

Besides these general descriptions of the business activities involved, from an organisational point of view, the following assumptions are made. Every customer should possess a shopping card, carrying information about the products and goods (s)he has opted for in a given shopping session. Payment is allowed only by bank credit-card, but this can be straightforwardly extended and adapted to other forms of payment such as bank transfers or cash. To boost competitiveness and thereby attract more customers, the existence of a customer management division is assumed. That is, as any e-commerce process or shopping session progresses, it is assumed that at least the full activity history of the customer is to be recorded and dynamically adapted, through the so-called customer profile. Such records should for instance include but not be limited to the amount of *spending* money, the *bill* for the purchased goods and the *date* of last update. Customers are asked to accomplish a sequence of succinctly predefined activities (for instance 'order', 'confirm' or 'buy') in order to obtain a product.

More precisely, the online shopping operation considered here distinguishes between normal, silver and gold customers. As an illustration, the following rules may apply for categorising customers. If a given customer has a history of heavy spending (e.g. greater than £3000), then (s)he is considered a gold customer, entitled to a 10% discount on any product, while those spending between £1000 and £3000 are ranked as silver customers and are offered a 5% discount. The profile of any customer depends thus on his/her spending history, which can be updated in two ways: (1) after the customer decides to pay, the total spent is increased according to the bill issued; (2) the updating of total spent is checked regularly. For example, if the spending is not updated after three months, the customer's spending history will be automatically reduced by a specific amount, such as £100. This may of course cause the customer's profile to change according to the rules above. These are just snapshots of rules which may govern any business activity. The purpose of this chapter is thus to explore a business activity and address its rules at the interaction level, following the three steps of the approach.

The following sections will incrementally apply the different steps of the approach with respect to each of these e-shopping business activities. That is, an informal description of the activity and its governing rules is first given, and then these informal rules are leveraged as ECA-driven architectural connectors. Next, associated Maude-based components and interfaces are considered and implemented, before the ECA-driven behavioural interactions

are specified and implemented in the extended Maude as aspects. Finally, the execution of such rules is dynamically woven into the respective components. For each activity, some rules have been selected which can be extended, enriched and evolved at any time. Moreover, concrete scenarios are executed in terms of selected component instances and rule agreements for each activity. This should show how any complex and realistic e-shopping application can be handled using the proposed approach.

5.2 The Developed Tool at Glance

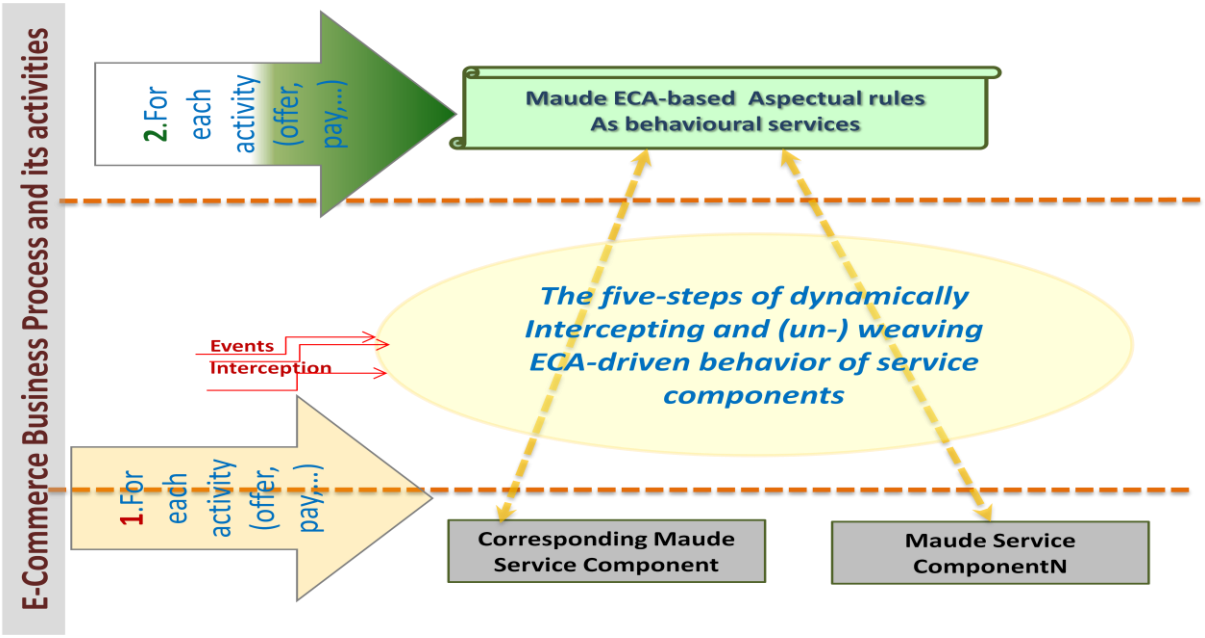


Figure 5.3: The tool main functionalities at glance

Before delving into the detailed conception, formal specification and dynamic adaptability of this service-oriented E-commerce application, it is necessary to first emphasis the main functionalities of the developed software tool. However, it is worth to mention that the tool itself concentrates on the automated steps, rather than the further conceptual steps that should be elaborated by the service designers. More precisely, as depicted in Figure 5.3, these main functionalities consist in the followings:

- Once the business activities composed the service-oriented business processes at-question have identified, the tool allows implementing all the involved service components for each activity using the extended Maude. In our case, the business activity product request for instance, involves the provider and customer service components. In this case, these components have to implement on the basis of their

informal descriptions. Service component instances are than implemented as running configuration.

- Once the required service components for any business activity are specified and then implemented and validated, the tool takes each ECA-driven descriptive rule and allows formally specifying it and then implementing it at the aspectual-level. The rules as well as the involved service component interfaces respect the developed patterns from the two previous chapters.

- The service instances which should be in contract or in interactions with each other are than defined at the aspectual configuration level.

- The next step consists in intercepting the triggering events for each activity, in the required causality-order according to the underlying business process. The tool, than applies the five-steps we developed in the previous chapter to dynamically adapt the behaviour of the invoked activity, depending on the corresponding rule at the aspectual level.

5.3 The Order Activity: The Approach at Work

First, the order activity involves the following business entities as services: the customer, the product and the shopping card. Informally, the governing intentional rule can be explained as follows. The customer orders specific products, through the triggering of event like *OrderPr (products, quantities).* When the order is possible, i.e. when the requested quantity is available, the product is ordered and added to the customer's basket.

5.3.1 The Order Activity (Phase-rule): The ECA-driven Governing Rule

In terms of a more refined ECA-driven operational rule, this order-governing rule can be straightforwardly presented in the following clearer form.

INTER-RULE: Order activity.
PARTNERS: Customer and Product and Shopping card.
EVENT: The customer orders specific quantities of particular products.
CONSTRAINTS: Checking that the shopping card belongs to the customer and that the quantity of product ordered is not greater than the available quantity.
ACTIONS: The list of products ordered and their requested quantities is added to the basket of the customer's shopping card.

5.3.2 The Order Activity (Phase-int): The ECA-driven Architectural Model

The above ECA-driven business rule governing the order activity clearly requires the following precise service interfaces from the participants (services). That is, as depicted on the right-hand side of Figure 5.4, first the product ordering should be triggered by the service interface of the customer through the event OrderPr, which includes the ordered product identity and the ordered quantity. From the product service component, the available quantity of the product is required. The shopping card service interface allows the ordered products to be added, through the associated message denoted AddPr2Bask.

The ECA-driven architectural interaction rule itself is depicted on the left-hand side of Figure 5.4, inspired by and mapped from the informal ECA-driven rule above. That is, on one hand, the event OrderPr is triggered by the customer service, then the constraints in the *under* clause have to be checked. These are that the shopping card should belong to the customer and that the quantity of the product ordered is not greater than its available quantity. Finally, this requested quantity of the ordered product is added to the customer's shopping card under the *acting* clause. As noted in the previous chapter, specific instance names have been used for the service interfaces involved to express precisely this architectural rule (e.g. Cust for customer, Prod for product and ShCd for shopping card). Note that the provider itself can still be added as an independent service or implicitly included in the product service (as here).

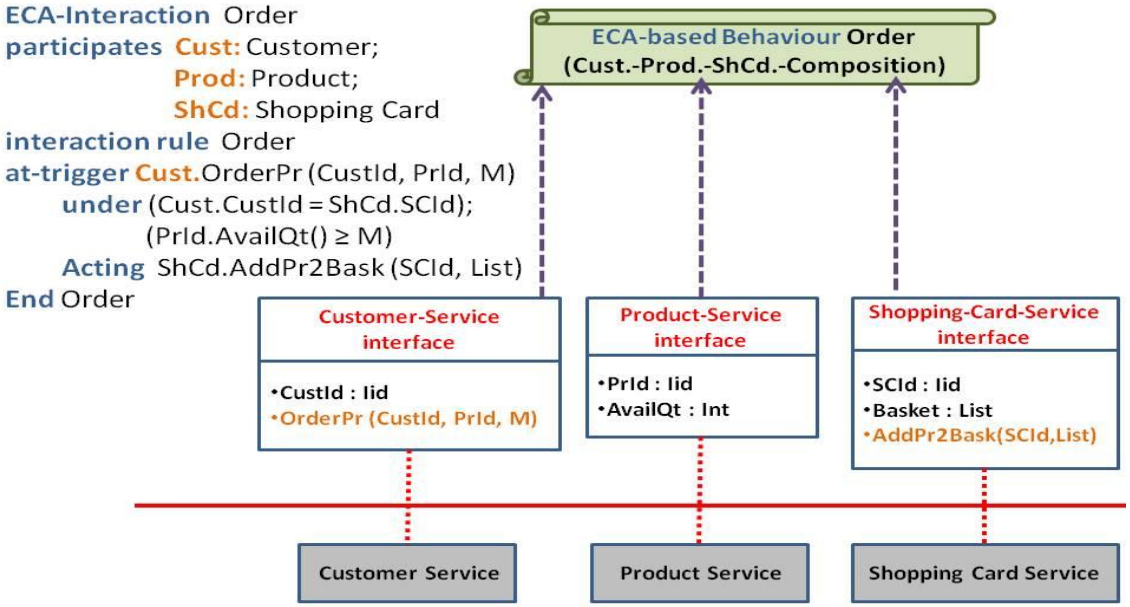


Figure 5.4: ECA-driven pattern of e-commerce for order interaction

5.3.3 The Order Activity (*Phase-Maude*): Aspectual Maude at Work

Since this phase is important and composed of at least three steps, it is extensively detailed as follows. This subsection begins by reporting on the specification of the service components involved, i.e. the customer, product and shopping card. Some features of their required service interfaces are then discussed, followed by a presentation of the Maude formalisation and validation of the respective ECA-driven rule. Finally, the runtime weaving of this rule into the relevant service component is addressed.

5.3.3.1 The Service Components Formalised and Validated in the Extended Maude

The complete specific service customer component with all imported modules, sorts, operations and variants is depicted in Appendix B, Figure B.1. The important elements of this service component have been extracted and are discussed in what follows. That is, each customer should have a name, address and profile as properties. The name and address are defined in lines 21 and 22 respectively as local properties and the customer can change his/her address through a local message chgAdr defined in line 20, the corresponding rule being defined in line 26. According to the above business rule, the customer can be designated gold, silver or normal. Therefore, Golden, Silver and Normal are defined as constants in this service customer component (line 11), with the profile as observed operation (line 23). Next, all possible messages or operations required for all different business activities are defined. That is, the customer service component should deliver all these messages to cover all business activities, and the adapted interface to a given activity will select the right messages. For the order business activity, for instance, the operation OrderPr (line 13) is needed. The others will be discussed as appropriate.

```
1. mod CUS_CMP is
. . .
11.ops  Normal Silver Golden: → Prof .
12.ops TakeYself Normal Express :→Shipment .
13.op OrderPr(_,_ ,_ ) :CustId PrId Int→ORDP .
14.op ask4Bill(_) : CustId → Ask4B .
15.op ask4Deliver(_,_) : CustId Shipment
        → Ask4D .
16.op ask4Pay(_) : CustId → Ask4P .
17.op succPay(_) : CustId → SuccP .
18.op getBill(_,_) : CustId Rat → GB .
19.op cancelPr(_,_,_) : CustId PrId Int
→ Cancel .
20.op ChgAdr(_,_) :CustId String →CHGADR .
21.op Name:_: String → loc Prop .
22.op Adr:_: String → loc Prop .
23.op Profile:_: Prof → obs Prop.
. . .
26.rl [chgAdr] : ChgAdr( CS, Anew )
      < CS | Adr: Aold > ⇒  < CS | Adr: Anew > .
```

In the same spirit, the product service component is defined below, as an extract of the complete code given in Appendix B, Figure B.2. That is, each service product should at least

be characterised by its available quantity and the price (lines 9-10) as observed properties. To update the product quantity, a corresponding observed message UpdatePr is defined (line 11). The corresponding implementation rule is defined in line 14 to change a given product with the updated quantity. Note again here that the business logic constraints are not present and are thus externalised as ECA-driven interaction rules.

```
1. mod PROD_CMP is                          . . .
. . .                                       14.rl[UpdatePr]:UpdatePr(Pid,M)<Pid | AvailQt:N>
9.op AvailQt:_: Nat → obs Prop .            15.        ⇒  < Pid | AvailQt: (N - M) > .
10.op Price:_: Int → obs Prop .             16. endm
11.op UpdatePr(_,_) :PrId Int→UPDPr .
```

Finally, the essential part of the shopping card service component is depicted as the following code, the complete specification being detailed in Appendix B, Figure B.3. The shopping card contains as observed properties the customer identity *CusId* (line 11) to which it belongs and the *basket* (line 12). The basket elements are in the form of a list consisting of product identities with ordered quantities (between lines 8-10). The ordered product can be added to the list of the basket through the observed message AddPr2Bask (line 13). The corresponding rule takes the form in line 18.

```
1. mod SHC_CMP is                           . . .
. . .                                       18. rl [AddPr2Bask] : AddPr2Bask(Cid, L1)
8. op [ ] : → List [ctor] .                 < Cid | Basket:L> ⇒ < Cid | Basket:(L ° L1)> .
9. op [ _,_ ] : PrId Int → List .           19. rl [CombineList] : [Pid, M] ° [Pid, N]
10. op _ ° _ : List List→ List [assoc comm].                ⇒ [Pid, M + N] .
11. op CusId:_ : CustId → obs Prop .
12. op Basket: _: List → obs Prop .         20. rl [DropNilList] : [ ] ° L ⇒ L .
13. op AddPr2Bask(_,_) : CdId List → ADP2B. 21. endm
```

5.3.3.2 The Required Service Interfaces Formalised and Validated in the Extended Maude

As analysed in the above ECA-driven architectural modelling rule for the order activity in Figure 6.3, three service interfaces are required: the customer, product and shopping card. For the customer service interface, the following important part has been extracted from the complete specification given in Appendix B, Figure B.9. The interface name CUST is first defined (line 8). The order product event OrderPr is intercepted through the rules getCfIntfOrdPr and getCfIntfOrdPrf (lines 18-22). Through the first rule getCfIntfOrdPr,

all the OrderPr events for those customer instances participating in any agreement at the interaction level are collected, while through the second rule getCfIntfOrdPrf, all these collected events are intercepted and passed to the interface state. Note that this agreement list *CusL* is dynamically passed from the interaction level when performing the interception using the strategy.

```
1.  mod CUS_INTF4ORD_GNR is
. . .
8.  op CUST : → Intf NM .
9.  op OrderPr(_,_,_) : CustId PrId Int → ORDP .
. . .
18.  rl [getCfIntfOrdPr] : getCfIntfOrdPr(OrderPr(CS, Pr, M) Cfcp, CusL, PrL)
19.        ⇒  (if (belong(CS, CusL) and belong(Pr, PrL))
20.        then OrderPr(CS, Pr, M) getCfIntfOrdPr(Cfcp, CusL, PrL)
21.        else getCfIntfOrdPr(Cfcp, CusL, PrL) fi) .
22.  rl [getCfIntfOrdPrf ]:getCfIntfOrdPrf(Cfcpf getCfIntfOrdPr(Cfcp, CusL, PrL)) ⇒  Cfcpf .
23. endm
```

The essential part of the service product interface for the order activity is extracted from the complete module in Appendix B, Figure B.7. Similarly to the customer interface, first a name (PROD) is given, then two rules are specified for intercepting the available quantity. The first rule getCfIntfAQ is used to collect the properties *AvailQt* from those product instances in agreement, that is, allowed to be ordered by such a customer. These collected properties are then intercepted through the second rule getCfIntfAQtf.

```
1. mod PROD_INTF4ORD_GNR is
. . .
7. op PROD : → Intf NM .
8. op AvailQt: _: Nat → obs_Prop .
.. .
16. rl [getCfIntfAQt] : getCfIntfAQt(< Pr | AvailQt: B > Cfcp, PrIdL)
17.     ⇒  (if belong(Pr, PrIdL)
18. then < Pr | AvailQt: B > getCfIntfAQt(Cfcp, PrIdL)
19. else     getCfIntfAQt(Cfcp, PrIdL) fi) .
20. rl [getCfIntfAQtf ] : getCfIntfAQtf(Cfcpf getCfIntfAQt(Cfcp, PrIdL))   ⇒  Cfcpf .
21. endm
```

Finally, the shopping card service interface for the order activity is to be formalised. Below is the main part extracted from the complete specification given in Appendix B, Figure B.8.

The interface name is defined as SHC in line 11, and then the properties CusId and *Basket* are intercepted through the rules getCfIntfBasket and getCfIntfBasketf (lines 26-30). The rules pendAddPr2Bask and pendAddPr2Baskf are further defined in lines 31 and 33 to pend the existing AddPr2Bask messages in the shopping card, at the start of the order activity. These pended messages should be activated again in the shopping card service component instance after the order activity, using the rule *backAddPr2Bask* in line 34. This prevents products being added directly to the shopping card service without passing through the order.

```
1. mod SHC_INTF4ORD_GNR is
• • •
11. op SHC : → Intf_NM .
12. op CusId: _: CustId → obs_Prop .
13. op Basket:_ : List → obs_Prop .
14. op AddPr2Bask( _,_ ) : CdId List → ADP2B.
• • •
26. rl [getCfIntfBasket]:getCfIntfBasket(<Cid|CusId: Cus> <Cid|Basket: L> Cfcp, CIdL)
27.      ⇒ (if belong(Cid, CIdL)
28.      then < Cid | CusId: Cus > < Cid | Basket: L > getCfIntfBasket(Cfcp, CIdL)
29.      else getCfIntfBasket(Cfcp, CIdL) fi) .
30. rl [getCfIntfBasketf ]:getCfIntfBasketf(Cfcpf getCfIntfBasket(Cfcp, CIdL)) ⇒ Cfcpf .
31. rl [pendAddPr2Bask]: pendAddPr2Bask(AddPr2Bask(Cid, L) Cfcp)
32.              ⇒ AddPr2Bask (Cid, L) pendAddPr2Bask (Cfcp) .
33. rl [pendAddPr2Baskf ]: pendAddPr2Baskf(Cfcpf pendAddPr2Bask(Cfcp)) ⇒ Cfcpf .
34. rl [backAddPr2Bask]: backAddPr2Bask(Cfcpf, Cfcp) ⇒ Cfcpf Cfcp .
35. endm
```

5.3.3.3 The Order ECA-driven Rule Formalised and Validated in the Extended Maude

Since the generic Maude-based ECA-driven interaction model has been introduced in the previous chapter, this paragraph concerns the direct mapping of the ECA-driven order rule presented in section 5.1 onto that extended Maude. The essential formalisation of the rule is given below, while full details are given in Appendix B, Figure B.10. The interaction name ORDER is first defined in line 7, and then *CS* is chosen for the customer, *Pr* for the product and *Cd* for the shopping card as service interface instances. In the *ORDER* interaction rule, defined in lines 11-16, the OrderPr event is sent from a partner *CS* and intercepted via the customer interface *CUST*. The property AvailQt is selected from the partner *Pr*, provided via the interface *PROD*. The partner *Cd* in its turn gives the basket information *Basket* and *CusId* properties, intercepted via the interface *SHC*. Under the available quantity as an

externalised constraint, the resulting behaviour is as follows. The OrderPr event is consumed (i.e. [CUST | nil]) and an AddPr2Bask message is sent to that shopping card (i.e. [*SHC* | *AddPr2Bask* (*Cd*, [*Pr, M]*) < *Cd*|*Basket*: *L* >]).

```
1. mod COORD_ORDER is
• • •
7. op ORDER : → Coord_NM .
•• •
11. crl [ORDER] : [ORDER || CS Pr Cd] & [CUST | OrderPr(CS, Pr, M)]
12.               & [PROD | < Pr | AvailQt: B >] & [SHC | < Cd | Basket: L >]
13.               & [SHC | < Cd | CusId: CS >]
14.       ⇒  [ORDER || CS Pr Cd] & [SHC | AddPr2Bask(Cd, [Pr, M]) < Cd | Basket: L >]
15.       & [SHC | < Cd | CusId: CS >] & [PROD | < Pr | AvailQt: B >] & [CUST | nil]
16.       if B >= M .
17. endm
```

5.3.3.4 The ECA-driven Order Rule and Dynamic weaving using the extended Maude

This order activity follows the proposed guiding steps on how to write strategy for intercepting and propagating events and required information from service components to the interaction level, then performing and dynamically weaving the result into such participating service components. The following Maude module depicts the essential elements of the strategy for non-intrusively performing the above ECA-interaction rule associated with that order activity. The complete strategy is shown in Appendix B, Figure B.11.

First, the SplitAT rule (lines 16-17) is applied to all participating service component configurations, namely the customer, product and shopping card. The rules pendAddPr2Bask and pendAddPr2Baskf are then implemented from lines 18 to 21, to pend all messages such as *AddPr2Bask*, as explained above. Next, the so-called interception begins from lines 22 to 41. The belong rule (lines 22-23) is used to extract just the state parts which are in agreement by using the agreed list of entity instances (i.e. concrete customers, products and shopping cards as illustrated). All order product events *OrderPr* are then intercepted from the customer service state through the rules getCfIntfOrdPr and getCfIntfOrdPrf (lines 24-27), while the rules getCfIntfAQt and getCfIntfAQtf intercept available quantities from the product service. All these intercepted elements are thus removed from corresponding service components (lines 40-41). These customer, product and shopping card interface states are first prepared to the term structure of the interaction, using

the rule Subsume (lines 42-43), then merged in the interaction state after being split, through the rule Split_CfIntf (lines 44-45). At this stage, the ORDER interaction rule is to be performed in lines 46-47. Once these rules have been executed at the interaction level, the next step is to recombine any split parts of different resulting interfaces, using the rule Recombin_CfIntf (lines 51-52). The different interface states are then separated using the rule extractCfIntf (lines 53-54). Finally, they are woven into the respective service components, using the rule weaveCfIntf from lines 55 to 63. The rules associated with different actions are to be performed at that service component level, such as the rule AddPr2Bask. After that, the resulting states are woven into the respective service components. Lastly, messages like AddPr2Bask are activated again through rule backAddPr2Bask (lines 63-64). At the end, any empty elements should be cleaned and all split elements recombined, using the rules RemoveNil and RecombineAT.

```
1. mod ASP_ORDER_Str is
. . .
13.op Compute : Term Nat → Term .
14.ceq Compute(T, N)
15. = if(N == 1) then
16.(if(SplitAT? ::Result4Tuple)
17.then Compute(getTerm(SplitAT?), N)
18.else if(pendAddPr2Bask? ::Result4Tuple)
19.then Compute(getTerm(pendAddPr2Bask?),N)
20.else if(pendAddPr2Baskf? ::Result4Tuple)
21.thenCompute(getTerm(pendAddPr2Baskf?),N)
22.else if(belong? :: Result4Tuple)
23.then Compute(getTerm(belong?),N)
24.else if(getCfIntfOrdPr?::Result4Tuple)
25.then Compute(getTerm(getCfIntfOrdPr?),N)
26.else if(getCfIntfOrdPrf?::Result4Tuple)
27.then Compute(getTerm(getCfIntfOrdPrf?),N)
28.else if(getCfIntfAQt? :: Result4Tuple)
29.then Compute(getTerm(getCfIntfAQt?),N)
30.else if(getCfIntfAQtf? :: Result4Tuple)
31.then Compute(getTerm(getCfIntfAQtf?),N)
32.else if(getCfIntfCus?::Result4Tuple)
33.then Compute(getTerm(getCfIntfCus?),N)
34.else if(getCfIntfCusf?::Result4Tuple)
35.then Compute(getTerm(getCfIntfCusf?),N)
36.else if(getCfIntfBasket?::Result4Tuple)
37.then Compute(getTerm(getCfIntfBasket?),N)
38.else if(getCfIntfBasketf?::Result4Tuple)
39.then Compute(getTerm(getCfIntfBasketf?),N)
40.else if(intercept? :: Result4Tuple)
41.then Compute(getTerm(intercept?),N)
42.else if(Subsume? :: Result4Tuple)
43.then Compute(getTerm(Subsume?), N)
44.else if(Split CfIntf?::Result4Tuple)
45.then Compute(getTerm(Split CfIntf?),N)
46.else if(ORDER? :: Result4Tuple)
47.then Compute(getTerm(ORDER?), N)
.. .
50.else
51.(if(Recombin CfIntf?::Result4Tuple)
52.then Compute(getTerm(Recombin CfIntf?),0)
53.else if(extractCfIntf?::Result4Tuple)
54.then Compute(getTerm(extractCfIntf?),0)
55.else if(weaveCfIntf? ::Result4Tuple)
56.then (if(AddPr2Bask? ::Result4Tuple)
57. then Compute(getTerm(AddPr2Bask?),0)
. . .
62.else Compute(getTerm(weaveCfIntf?),0))
63.else if(backAddPr2Bask?::Result4Tuple)
64.then Compute(getTerm(backAddPr2Bask?),0)
65.else if(RemoveNil? :: Result4Tuple)
66.then Compute(getTerm(RemoveNil?),0)
67.else if(RecombineAT? :: Result4Tuple)
68.then Compute(getTerm(RecombineAT?),0)
. . .
```

5.4 The Confirmation Activity: The Approach at Work

5.4.1 The Confirmation Activity (*Phases-rule+Int*): The ECA-driven Rule and its Architectural Modelling

After adding the ordered products into their shopping cards, customers are requested to engage in the confirmation activity. It is assumed for simplicity that they have systematically accepted the offer. In this activity, whose service components are customer, product, shopping card and history, the total customer-profiled bill for the products ordered so far is calculated. Since each customer has a profile (e.g. normal, silver or gold), different discount rates will influence the total bill. The informal description of the ECA-driven business rule for this activity is as follows:

COORD-RULE: Confirmation activity.
PARTNERS: Customer and Product and Shopping card and History.
ATTRIBUTE: Discount rate.
EVENT: The customer asks for bill.
CONSTRAINTS: Checking the available quantity of the products.
ACTIONS: The bill for the products in the basket is calculated according to the quantity and the price of the product; the actual bill is calculated considering the discount rate and recorded history; the customer receives the actual bill.

According to the above informally described ECA-driven rule, the respective precise interfaces from the partners involved are to be produced for this activity. That is, for this confirmation activity, the following service interfaces[2] have to be selected: the service customer interface *CustCFMSI*, the product service interface *ProdCFMSI*, the shopping card interface *ShCCFMSI* and the history service interface *HisCFMSI*. The corresponding ECA-driven architecture model is depicted on the right-hand side of Figure 5.5.

More precisely, the confirmation activity is first triggered by the customer through the event *ask4Bill*. This event is to be intercepted from the customer service component. In the product service interface, the price and available quantity of all of requested products are considered. The observed message *UpdatePr* (*pr*, *Qt*) allows the given product to be updated with the requested quantity. In the shopping card service interface, the properties *CusId* and *Basket* are considered. The observed property *bill* is required to record the bill for the products

[2] Note that there is no need to formally specify any of these service components, as it has already been done. The required service interfaces are thus selected directly, except for the history service component, which is used here for the first time.

ordered by the customer. Hence, from the history service interface the properties *CusId* and *bill* are required.

The ECA-driven architectural interaction rule, depicted on the left-hand side of Figure 5.5, includes the four interfaces above, while using instance names for them. As discussed already, customers have discount rates depending on their profiles, so that for instance, those with a gold profile are entitled to a 10% discount, those with a silver profile 5% and others none. Therefore, the different discounts rate—*NormalRd*, *SilverRd*, and *GoldenRd*—are defined as constants at the interaction level. The discount rate attribute *discrete* is defined to make the business rule more flexible.

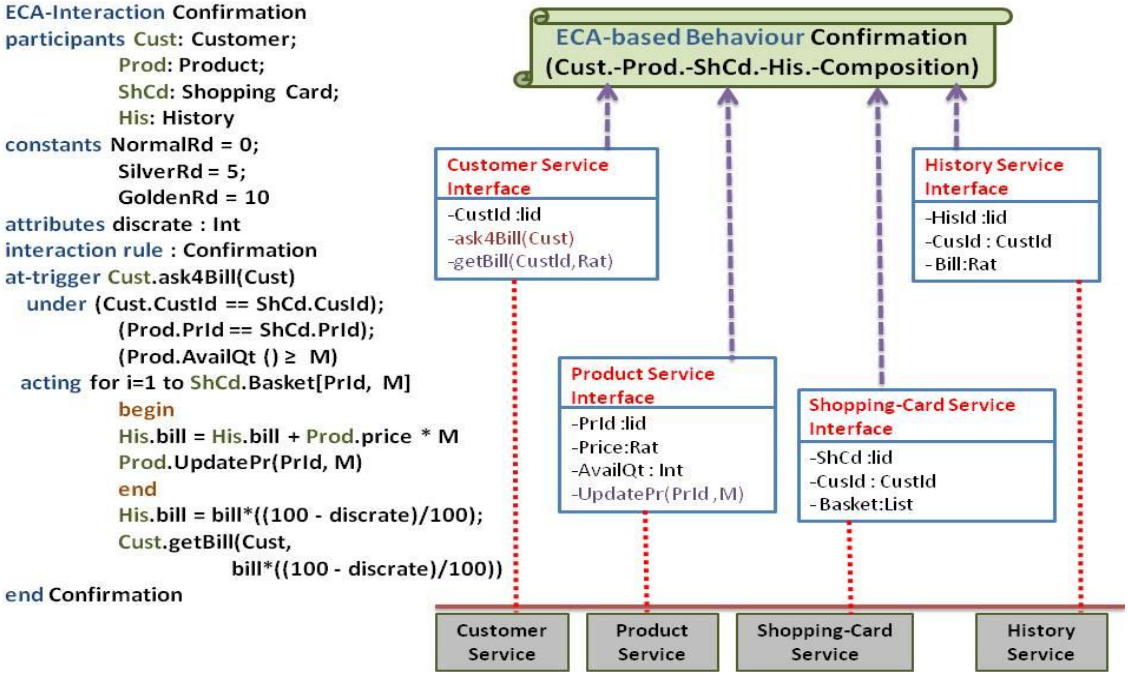


Figure 5.5: ECA-driven pattern of e-commerce for confirmation interactions

The ECA-driven interaction rule is thus triggered by the customer through the event *ask4Bill*. After checking the available quantity, the bill for the products in the customer's basket is calculated according to the quantities ordered and current prices. At the same time, the available quantities for requested products are updated. After that, the computed bill with the discount rate is recorded in the history and sent to the customer.

5.4.2 The Confirmation Activity (*Phase-Maude*): Aspectual Maude at Work

5.4.2.1 The Service Components Formalised and Validated in the Extended Maude

Since the Maude formalisation for the service components customer, product and shopping card have already been introduced in relation to the order activity, only the history service component is presented here. As depicted below, this includes the following properties (lines 11-14): customer identity *CusId*, spending records *Spending*, date of last spending update *LastUpdateSP* and the current *bill* for products ordered. An observed message *UpdateSP* is used to update customer spending (lines 20-21). Note that the history receives the current day automatically through the message *getCurrentDay* in line 16. Since dates and differences between dates are needed, the complete specification in appendix B, Figure B.4 also defines a date data-type, which is imported here.

```
1. mod HIS_CMP is
. . .
11.op CusId:_ : CustId → obs_ Prop .
12.op Spending: _: Rat → obs_ Prop .
13.op LastUpdateSP: _: Date → obs_ Prop .
14.op bill: _: Rat → obs _Prop .
15.op UpdateSp(_,_) : CustId Rat → UPDSP .
16.op getCurrentDay(_,_) : HisId Date → GCDAY [ctor] .
. . .
20.rl [UpdateSp]: UpdateSp(CId, M)< Hid | CusId: CId, Spending: S >
21. ⇒ <Hid | CusId: CId, Spending:(S + M)> .
```

5.4.2.2 The Required Service Interfaces Formalised and Validated in the Extended Maude

First, as the confirmation activity is triggered by the customer, the customer service interface is described in the following Maude module. The rules getCfIntfAsk4B and getCfIntfAsk4Bf (lines 20-24) are thus defined as permitting the interception of the event *ask4Bill* and the property *Profile*, for agreed partner instances at the interaction level. the complete specification of the customer service interface is depicted in appendix B, Figure B.12.

```
1. mod CUS_INTF4CFM_GNR is
. . .
8. op CUST : → Intf_ NM .
9. op ask4Bill(_) : CustId → Ask4B .
10. op getBill(_,_) : CustId Rat → GB .
11. ops Normal Silver Golden : → Prof .
12. op Profile: _: Prof → obs_ Prop .
. . .
20. rl [getCfIntfAsk4B] : getCfIntfAsk4B(ask4Bill(CS) < CS | Profile: prf > Cfcp, CusL)
21.        ⇒ (if belong(CS, CusL)
```

```
22.     then ask4Bill(CS) < CS | Profile: prf > getCfIntfAsk4B(Cfcp, CusL)
23.     else getCfIntfAsk4B(Cfcp, CusL) fi) .
24. rl [getCfIntfAsk4Bf ] : getCfIntfAsk4Bf(Cfcpf getCfIntfAsk4B(Cfcp, CusL)) ⇒ Cfcpf .
25. endm
```

The required product service interface is named below as INTF4CFM_GNR. The price and the available quantity of all products are thus required. These are intercepted through the rules getCfIntfAQt and getCfIntfAQtf (lines 19-21).

```
1. mod PROD_INTF4CFM_GNR is
. . .
8. op PROD : → Intf_ NM .
9. op AvailQt: _: Nat → obs_Prop.
10. op Price: _: Rat → obs_Prop.
11. op UpdatePr( _, _) : PrId Int → UPDPr[ctor].
. . .
19. rl [getCfIntfAQt] : getCfIntfAQt(< Pr | AvailQt: B > < Pr | Price: M > Cfcp)
20.     ⇒ < Pr | Price: M > < Pr | AvailQt: B > getCfIntfAQt (Cfcp).
21. rl [getCfIntfAQtf]: getCfIntfAQtf (Cfcpf getCfIntfAQt(Cfcp)) ⇒ Cfcpf .
22. endm
```

The Maude-based formalisation of the history service interface related to the confirmation activity is as given below. The rules getCfIntfSpd and getCfIntfSpdf in lines 18-22 intercept the required properties *CusId* and *bill* from the history service. The complete specification of history service interface is depicted in Appendix B, Figure B.15.

```
1. mod HIS_ INTF4CFM_ GNR is
. . .
7. op HIS : → Intf_NM .
8. op CusId: _: CustId → obs_Prop .
9. op bill: _: Rat → obs_Prop .
. . .
18. rl [getCfIntfSpd]: getCfIntfSpd(< His | CusId: Cus > < His | bill: B > Cfcp, HisL)
19.     ⇒ (if belong(His, HisL)
20.     then < His | CusId: Cus > < His | bill: B > getCfIntfSpd(Cfcp, HisL)
21.     else getCfIntfSpd(Cfcp, HisL) fi) .
22. rl [getCfIntfSpdf ] : getCfIntfSpdf(Cfcpf getCfIntfSpd(Cfcp, HisL)) ⇒ Cfcpf .
23. endm
```

Finally, in order to calculate the bill, the basket service interface is required. This includes all ordered products with respective quantities. As depicted below, the interception is performed through the rules getCfIntfBasket and getCfIntfBasketf (lines 21-26). In Appendix B, Figure B.14 the complete specification of that shopping card service interface is given.

```
1. mod SHC_INTF4CFM_GNR is
. . .
10. op SHC : → Intf_NM .
11. op CusId:_ : CustId → obs_Prop .
12. op Basket: _: List → obs_Prop .
. . .
21. rl [getCfIntfBasket] : getCfIntfBasket(< Cid | CusId: Cus >
22.       < Cid | Basket: L > Cfcp, CIdL)
23.       ⇒ (if belong(Cid, CIdL)
24.       then < Cid | CusId: Cus > < Cid | Basket: L > getCfIntfBasket(Cfcp, CIdL)
25.       else getCfIntfBasket(Cfcp, CIdL) fi) .
26. rl [getCfIntfBasketf ] : getCfIntfBasketf(Cfcpf getCfIntfBasket(Cfcp, CIdL)) ⇒ Cfcpf .
27. endm
```

5.4.2.3 The Confirmation ECA-driven Rule Formalised and Validated in the Extended Maude

Associated with the ECA-driven architectural interaction for the confirmation activity shown in Figure 6.4 is the corresponding Maude-based ECA-driven interaction rule. Line 11 names that interaction as CFM. The discount rates *discrete* are declared as an extra attribute in line 12. Three constants—*NormalRd*, *SilverRd* and *GoldenRd*— are also defined in lines 13-15 as discount rates for the three profiles of customers. As this activity is triggered by the customer through the event *ask4Bill*, the bill for all products ordered is calculated and the available quantities of these products are updated (lines 21-27). ECA-driven interaction rule *CFM* (lines 28 to 32) allows the customer to receive the bill, while taking into account any discount rates. That bill is further stored in his/her history. In Appendix B, Figure B.16 the complete specification of that confirmation service interface is given.

```
1. mod COORD_ CFM is
. . .
10. ops NormalRd SilverRd GoldenRd : → ProfRd
11. op CFM : → Coord_NM .
12. op discrate:_ : Int → Attribute [ctor gather (&)] .
13. eq NormalRd = 0 .
14. eq SilverRd = 5 .
```

```
15. eq GoldenRd = 10 .
. . .
21. crl [Calculate] : [PROD | < Pr | Price: P >] & [PROD | < Pr | AvailQt: Q >]
22.                 & [SHC | < Cd | Basket: ([Pr, M] L) >] & [SHC | < Cd | CusId: CS >]
23.                 & [HIS | < His | CusId: CS >] & [HIS | < His | bill: B >]
24.       ⇒  [PROD | < Pr | Price: P >] & [PROD | UpdatePr(Pr, M) < Pr | AvailQt: Q >]
25.       & [SHC | < Cd | Basket: L >] & [SHC | < Cd | CusId: CS >]
26.       & [HIS | < His | CusId: CS >] & [HIS | < His | bill: ((M * P) + B) >]
27.       if Q ≥ M .
28. rl [CFM] : [CFM || (CS $ Pr $ Cd $ His) @ discrate: R] & [CUST | ask4Bill(CS)]
29.                 & [HIS | < His | CusId: CS >] & [HIS | < His | bill: B >]
30.       ⇒  [CFM || (CS $ Pr $ Cd $ His) @ discrate: R] & [HIS | < His | CusId: CS >]
31.       & [CUST | getBill(CS, (B * ((100 - R) / 100)))]
32.       & [HIS | < His | bill: (B * ((100 - R) / 100)) >].
33. endm
```

5.4.2.4 The Confirmation ECA-driven Rule Formally and Dynamically Woven Using Maude

Through the rules getCfIntfAsk4B and getCfIntfAsk4Bf, all the trigger events *ask4Basket* are intercepted from the customer service state. Similarly, the rules getCfIntfAQt and getCfIntfAQtf select available quantities and prices from the product service. Through the rules getCfIntfBasket and getCfIntfBasketf, the shopping-card service interface state is intercepted. The bill is intercepted using the rules getCfIntfSpd and getCfIntfSpdf. Lines 38-39 prepare and merge these intercepted states with the interaction state. At this stage the CFM interaction rule is performed (lines 42 to 46). The result is then recombined by lines 50-51. Finally, they are woven into the respective service components using the rule weaveCfIntf (lines 54-58). In Appendix B, Figure B.17 the complete specification of that confirmation service interface is given.

```
1. mod ASP_CFM_Str is
. . .
13.op Compute : Term Nat → Term .
14.ceq Compute(T, N)
15.= if(N == 1) then
16.(if(SplitAT? :: Result4Tuple)
17.then Compute(getTerm(SplitAT?), N)
18.else if(belong? :: Result4Tuple)
19.then Compute(getTerm(belong?),N)
20.else if(getCfIntfAsk4B? :: Result4Tuple)
21.then Compute(getTerm(getCfIntfAsk4B?),N)
22.else if(getCfIntfAsk4Bf?::Result4Tuple)
23.then Compute(getTerm(getCfIntfAsk4Bf?),N)
24.else if(getCfIntfAQt? :: Result4Tuple)
36.else if(intercept? :: Result4Tuple)
37.then Compute(getTerm(intercept?),N)
38.else if(Subsume? :: Result4Tuple)
39.then Compute(getTerm(Subsume?), N)
40.else if(Split CfIntf?::Result4Tuple)
41.then Compute(getTerm(Split_CfIntf?),N)
42.else if(CFM? :: Result4Tuple)
43.then (if (Calculate?::Result4Tuple)
44. then Compute(getTerm(Calculate?),N)
45. else Compute(getTerm(CFM?), N)
. . .
40.else
50.(if (Recombin_CfIntf?::Result4Tuple)
51.then Compute(getTerm(Recombin_CfIntf?),0)
```

25.then Compute(getTerm(getCfIntfAQt?),N)	**52.else if** extractCfIntf?::Result4Tuple
26.else if(getCfIntfAQtf? :: Result4Tuple)	**53.then** Compute(getTerm(extractCfIntf?),0)
27.then Compute(getTerm(getCfIntfAQtf?),N)	**54.else if (**weaveCfIntf?::Result4Tuple)
28.else if(getCfIntfBasket?::Result4Tuple)	**55.then (if(**UpdatePr? :: Result4Tuple)
29.then Compute(getTerm(getCfIntfBasket?),N)	**56. then** Compute(getTerm(UpdatePr?),0)
30.else if(getCfIntfBasketf?::Result4Tuple)	**57. else** Compute(getTerm(weaveCfIntf?),0)
31.then Compute(getTerm(getCfIntfBasketf?),N)	**58. fi)**
32.else if(getCfIntfSpd? :: Result4Tuple)	**59.else if**(RemoveNil? :: Result4Tuple)
33.then Compute(getTerm(getCfIntfSpd?),N)	**60.then** Compute(getTerm(RemoveNil?),0)
34.else if(getCfIntfSpdf? :: Result4Tuple)	**61.else if(**RecombineAT? :: Result4Tuple)
35.then Compute(getTerm(getCfIntfSpdf?),N)	**62.then** Compute(getTerm(RecombineAT?),0)
	. . .

5.5 The Cancelling Activity: The Approach at Work

Normally, after confirmation, the customer should decide how to take delivery of the products. Nevertheless, for different reasons, in some cases the customer may want to cancel some products already ordered. This means that in the cancelling activity, the computed bill must be recalculated. The cancelling activity requests as services the customer, product and history. For each cancelation activity, the customer must pay a penalty, which in this case study may be as high as 20% of the bill.

5.5.1 The Cancelling Activity (Phases-rule+Int): The ECA-driven Rule and its Architectural Modelling

As mentioned above, the corresponding ECA-driven interaction cancellation rule requires the following service interfaces, as depicted on the right-hand side of Figure 5.6. From the customer *CustCancelSI*, the event *cancelPr* (*cs*, *pr*, *m*) is requested. During the cancellation activity, the current bill for remaining confirmed products must be recalculated, which means that the event *getBill* will have to be requested again.

```
ECA-Interaction Cancel
participants Cust: Customer;
             Prod: Product;
             His: History
constants NormalRd = 0;
             SilverRd = 5;
             GoldenRd = 10;
             pebalty = 20
attributes discrate : Int;
             Penalty : Int
interaction rule : Cancel
at-trigger Cust. cancelPr(Cust, PrId, M)
  under (His.bill)≥
     ((PrId.price() * M) * ((100 - discrate)/100))
  acting His.bill = His.bill -((PrId.price()* M)
       *((100 - discrate)/ 100));
    Cust.getBill(Cust, (bill - ((Prod.price()* M)
     * ((100 - discrate - penalty)/100))));
         Prod.UpdatePr(PrId, -M)

end Cancel
```

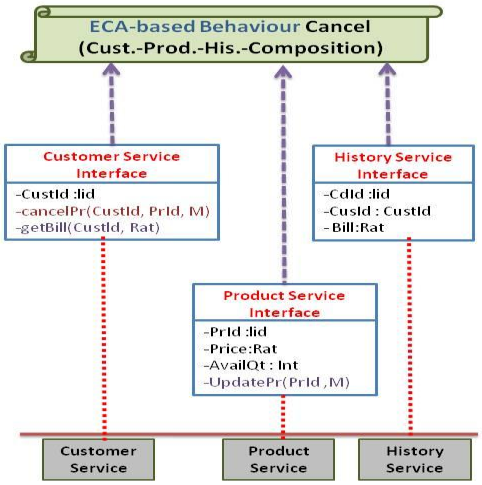


Figure 5.6: ECA-driven pattern of e-commerce for cancellation interactions

The available quantity of the cancelled products will then be updated. As depicted in the product service interface *ProdCancelSI*, the properties *AvailQt* and *Price* and the message *UpdatePr* (*PrId, M*) are therefore required. Upon recalculating the bill, the actual bill recorded in the history must be changed. Therefore, the bill information is required in the history interface *HisCancelSI*.

On the left-hand side of Figure 5.6 the specific interaction rule is formalised. The customer can cancel a specific number of confirmed products by repeatedly triggering the event *cancelPr*. As the bill has to be recalculated, the discount rates for customers are thus needed. The customer is then "punished" with a financial penalty, set for simplicity here at 20% of the total bill. Therefore, two extra attributes are defined: *discrete* and *Penalty*. The customer receives a recalculated bill taking into account the penalty.

5.5.2 The Cancelling Activity (*Phase-Maude*): Aspectual Maude at Work

As noted above, the behavioural ECA-driven interaction rule of the cancellation business activity requires the service interfaces customer, product and history. Fortunately, all of these have already been requested in the previous activities (e.g. order and confirmation), so they have already been formalised and the required service interfaces can now be directly extracted from them. Once again, the complete specification, execution and validation of these service components are given in Appendix B section B.1.3.

5.5.2.1 Service Iinterfaces: Formalisation and Validation in Extended Maude

As shown in the extract below, the customer service interface, formalised from the above cancellation rule, is named *CUS_INTF4CANCEL_GNR*. The complete specification of this module is given in Appendix B, Figure B.18. This essential extract shows the events *cancelPr* and *getBill* for triggering the cancellation activity. The corresponding rules getCfIntfCancel and getCfIntfCancelf (line 20-24) allow these two events to be intercepted from the customer service component.

```
1. mod CUS_INTF4CANCEL_GNR is
• • •
8. op CUST : → Intf_ NM .
9. op cancelPr( _,_ , _) : CustId PrId Int → Cancel .
10. op getBill( _,_ ) : CustId_Rat → GB .
• • •
20. rl [getCfIntfCancel] : getCfIntfCancel(getBill(CS, B) cancelPr(CS, Pr, M) Cfcp, CusL)
```

```
21.          ⇒ (if belong(CS, CusL)
22.          then cancelPr(CS, Pr, M) getBill(CS, B) getCfIntfCancel(Cfcp, CusL)
23.          else getCfIntfCancel(Cfcp, CusL) fi) .
24. rl [getCfIntfCancelf ] : getCfIntfCancelf(Cfcpf getCfIntfCancel(Cfcp, CusL)) ⇒ Cfcpf .
25. endm
```

The specific Maude-based product service interface *PROD_INTF4CANCEL_GNR* implemented in Maude Workstation is shown in Appendix B, Figure B.19 and the following essential extract explains it. The price and the available quantity of all the products are intercepted through the getCfIntfAQt and getCfIntfAQtf rules (lines 19-21). As defined in the corresponding ECA-driven interaction rule, these two product properties are required to recompute the bill, while cancelling the selected products.

```
1. mod PROD_INTF4CANCEL_GNR is
• • •
8. op PROD : → Intf_ NM .
9. op AvailQt:_ : Nat → obs_Prop .
10. op Price:_ : Rat → obs_Prop .
11. op UpdatePr( _, _) : PrId Int → UPDPr .
• • •
19. rl [getCfIntfAQt] : getCfIntfAQt(< Pr | AvailQt: B > < Pr | Price: M > Cfcp)
20.          ⇒ < Pr | Price: M > < Pr | AvailQt: B > getCfIntfAQt (Cfcp).
21. rl [getCfIntfAQtf ] : getCfIntfAQtf(Cfcpf getCfIntfAQt(Cfcp)) ⇒ Cfcpf .
22. endm
```

Finally, from the history service component the formal interface *HIS_INTF4CANCEL_GNR* is required. It is specified fully in Appendix B, Figure B.20. The following extract includes the rules getCfIntfSpd and getCfIntfSpdf (lines 18-22), which allow as usual the interception of the required properties from that history service, namely *CusId* and *bill*.

```
1. mod HIS_INTF4CANCEL_GNR is
• • •
7. op HIS : → Intf_NM .
8. op CusId: _: CustId → obs_Prop .
9. op bill:_ : Rat → obs_Prop .
• • •
18. rl [getCfIntfSpd]: getCfIntfSpd(< His | CusId: Cus > < His | bill: B > Cfcp, HisL)
19.          ⇒ (if belong(His, HisL)
20.          then < His | CusId: Cus > < His | bill: B > getCfIntfSpd(Cfcp, HisL)
21.          else getCfIntfSpd(Cfcp, HisL) fi) .
22. rl [getCfIntfSpdf ] : getCfIntfSpdf(Cfcpf getCfIntfSpd(Cfcp, HisL)) ⇒ Cfcpf .
23. endm
```

5.5.2.2 The ECA-driven Cancel Rule: Formalisation and Validation in Extended Maude

Using these service interfaces, the corresponding Maude-based formalisation of the above ECA-driven interaction for the cancellation activity is partially specified as follows, the complete formalisation and validation being given in Appendix B, Figure B.21. The name of this interaction rule is CANCEL (line 11). As already described, two extra attributes are required: *discrete* and *penalty* (lines 12-13). The ECA-driven interaction rule CANCEL, which is formalised between lines 21 and 35, allows the customer to cancel any confirmed products. This requires the recalculation of the running bill as well as an update of the quantity of any cancelled product. At this refined and precise level, it must be checked whether all products have been cancelled, making the recalculated bill less than zero. In such cases, the bill is not recorded as a negative quantity, but as zero.

```
1. mod COORD_CANCEL is
. . .
9. ops NormalRd SilverRd GoldenRd : → ProfRd .
10. op penalty : → Penalty .
11. op CANCEL : → Coord_ NM .
12. op discrate: _: Int → Attribute [ctor gather (&)] .
13. op Penalty:_ : Int → Attribute [ctor gather (&)] .
14. eq NormalRd = 0 .
15. eq SilverRd = 5 .
16. eq GoldenRd = 10 .
17. eq penalty = 20 .
. . .
21. rl [CANCEL] : [CANCEL || (CS $ Pr $ His) @ ((discrate: R); (Penalty: penal))]
22.                & [CUST | cancelPr(CS, Pr, M)] & [CUST | getBill(CS, B1)]
23.                & [PROD | < Pr | Price: P >] & [PROD | < Pr | AvailQt: Q >]
24.                & [HIS | < His | CusId: CS >] & [HIS | < His | bill: B >]
25.       ⇒  if((B - ((M * P) * ((100 - R) / 100))) > 0)
26.        then [CANCEL || (CS $ Pr $ His) @ ((discrate: R); (Penalty: penal))]
27.                & [CUST | getBill(CS, (B1 - ((M * P) * ((100 - R - penal) / 100))))]
28.                & [PROD | UpdatePr(Pr, (- M)) < Pr | AvailQt: Q >]
29.                & [PROD | < Pr | Price: P >] & [HIS | < His | CusId: CS >]
30.                & [HIS | < His | bill: (B - ((M * P) * ((100 - R) / 100))) >]
31.       else [CANCEL || (CS $ Pr $ His) @ ((discrate: R); (Penalty: penal))]
32.                & [CUST | getBill(CS, (B * (penal / 100)))] & [PROD | < Pr | Price: P >]
33.                & [PROD | UpdatePr(Pr, (- M)) < Pr | AvailQt: Q >]
34.                & [HIS | < His | CusId: CS >] & [HIS | < His | bill: 0 >]
35.       fi .
36. endm
```

5.5.2.3 The Dynamic Weaving Of the Cancellation Rule Using Extended Maude

The cancellation process is controlled by the strategy *ASP_CANCEL_Str*, which is fully detailed in Appendix B, Figure B.22. The first step in the essential extract below is to intercept the events *cancelPr* and *getBill* from the customer service state. The rules getCfIntfAQt and getCfIntfAQtf are then enacted to capture the available quantity *AvailQt* and price *Price* from the product service. The rules getCfIntfSpd and getCfIntfSpdf allow the properties *CusId* and *bill* to be captured from the history service.

These intercepted service interface states are prepared and merged with the corresponding ECA-driven interaction state, using the rule Subsume (lines 34-36). Next, the above CANCEL interaction rule (line 38) is enacted and any split parts of that result recombined using Recombin_CfIntf (line 43). The result of that interaction rule is then dispatched to different service interfaces using the rule extractCfIntf (line 45). Finally, these are dynamically woven into the respective service components via weaveCfIntf (lines 46-50). The rules associated with different actions are to be invoked at the service component level, such as the rule UpdatePr in line 48. At the end, any empty elements are cleaned and all split elements recombined through the rules RemoveNil and RecombineAT.

```
1. mod ASP_CANCEL_Str is
. . .
12.op Compute : Term Nat → Term .
13.ceq Compute(T, N)
14.= if(N == 1) then
15.(if(SplitAT? :: Result4Tuple)
16.then Compute(getTerm(SplitAT?), N)
17.else if(belong? :: Result4Tuple)
18.then Compute(getTerm(belong?),N)
19.else if(getCfIntfCancel?::Result4Tuple)
20.then Compute(getTerm(getCfIntfCancel?),N)
21.else if(getCfIntfCancelf?::Result4Tuple)
22.then Compute(getTerm(getCfIntfCancelf?),N)
23.else if(getCfIntfAQt? :: Result4Tuple)
24.then Compute(getTerm(getCfIntfAQt?),N)
25.else if(getCfIntfAQtf? :: Result4Tuple)
26.then Compute(getTerm(getCfIntfAQtf?),N)
27.else if(getCfIntfSpd? :: Result4Tuple)
28.then Compute(getTerm(getCfIntfSpd?),N)
29.else if(getCfIntfSpdf?::Result4Tuple)
30.then Compute(getTerm(getCfIntfSpdf?),N)
31.else if(intercept? :: Result4Tuple)
32.then Compute(getTerm(intercept?),N)
33.else if(Subsume? :: Result4Tuple)
34.then Compute(getTerm(Subsume?), N)
35.else if(Split_CfIntf?::Result4Tuple)
36.then Compute(getTerm(Split_CfIntf?),N)
37.else if(CANCEL? :: Result4Tuple)
38.then Compute(getTerm(CANCEL?), N)
39.else Compute(T, 0)
. . .
41.else
42.(if(Recombin CfIntf?::Result4Tuple)
43.then Compute(getTerm(Recombin CfIntf?),0)
44.else if(extractCfIntf?::Result4Tuple)
45.then Compute(getTerm(extractCfIntf?),0)
46.else if (weaveCfIntf?::Result4Tuple)
47.then (if(UpdatePr? :: Result4Tuple)
48. then Compute(getTerm(UpdatePr?),0)
49. else Compute(getTerm(weaveCfIntf?),0)
50. fi)
51.else if(RemoveNil? :: Result4Tuple)
52.then Compute(getTerm(RemoveNil?),0)
53.else if(RecombineAT? :: Result4Tuple)
54.then Compute(getTerm(RecombineAT?),0)
 . . .
```

5.6 The Shipment Activity: The Approach at Work

In this business activity customers choose how to receive their products. Shipment companies have several alternatives. For instance, products can be delivered by air, by sea or by land. Shipment cost may also depend generally on weight, distance, time, volume and so on. Customers should have the ability to choose any appropriate service that suits them in terms of cost and time (express, fast or normal delivery). In order to allow full flexibility and adaptability, shipment services must interact with both provider and customer. The resulting negotiation represents the core of the corresponding business logic of that activity and at best should be governed by appropriate ECA-driven business rules to promote abstraction, flexibility and dynamic adaptability.

5.6.1 The Shipment Business Activity (*Phases-rule+Int*): The ECA-driven Rule and its Architectural Modelling

As with cancellation, this paragraph directly specifies the ECA-driven architectural interaction rule. Figure 5.7 show that the associated services (services and interfaces) for that shipment activity include the customer, shipment provider, product provider and history. As graphically illustrated on the right-hand side, the event *ask4Deliver* (*CustId, Ship*) is required from the customer service interface *CustShipSI* to initiate any delivery. The *bill* is also required from the customer, so that it can be recalculated by adding the shipment cost. The shipment provider must provide at least the address and the shipment rate (skipped here for simplicity). The provider's address (at least) is also needed, as is the history, to reward customers opting for specific shipments (express, for instance).

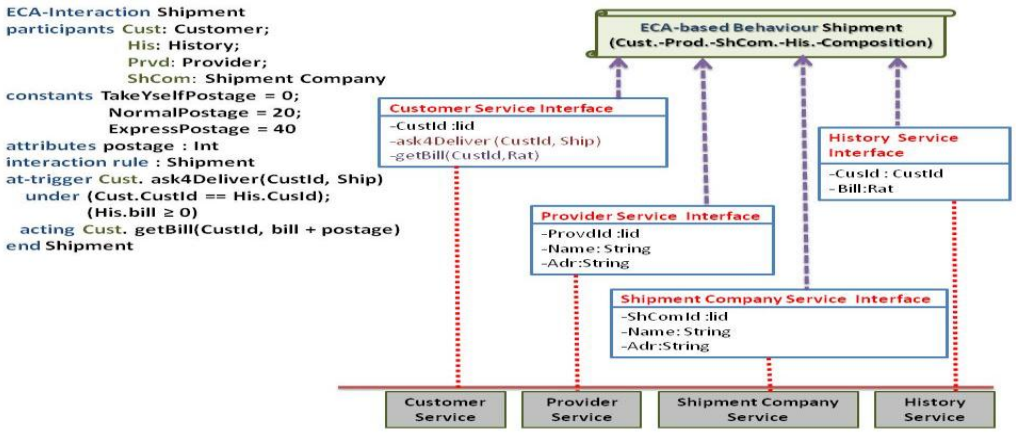


Figure 5.7: ECA-driven pattern of e-commerce for shipment interactions

The left-hand side of Figure 5.7 shows an illustrative basic shipment variant; other more complex ones could be added. More precisely, it is proposed that customers can obtain products in three ways: (1) collecting themselves, in which case the shipment cost *TakeY_selfPostage* is zero; (2) delivery by normal post, taking more time, but costing, for instance,[3] only £20; (3) the Express service, which is faster and safer and costs £40. To make such shipment rules more flexible, cost is defined as an extra attribute in the interaction rule, not within the shipment service. Customers ask for a specific form of delivery through the event *ask4Deliver*. If the current bill is greater than zero (i.e. (s)he has already confirmed at least one product), the customer's bill is incremented with the shipment cost.

5.6.2 The Shipment Activity (*Phase-Maude*): Aspectual Maude at Work

The Maude-based formalisation of the service components required for that shipment activity is already given. That is, while the customer and history services have been specified and used in previous activities, the provider and shipment services are considered trivial and are skipped here. However, they are fully specified and validated in appendix B, section B.1.4.

5.6.2.1 Service Interfaces: Formalisation and Validation in the Extended Maude

According to the simplified ECA-driven architectural interaction rule for the shipment activity modelled above, the events *ask4Deliver* and *getBill* are to be intercepted from the customer service component which given in Appendix B, Figure B.23. In the following Maude-based customer service interface, two rules, getCfIntfAsf4D and getCfIntfAsf4Df (lines 21-25), are defined for that purpose.

```
1. mod CUS_INTF4SHIP_GNR is
• • •
8. op CUST : → Intf_NM .
9. op TakeYself Normal Express : → Shipment .
10. op getBill(_,_) : CustId Rat → GB .
11. op ask4Deliver(_,_) : CustId Shipment → Ask4D .
• • •
20.rl [getCfIntfAsk4D]: getCfIntfAsk4D(getBill(CS, B) ask4Deliver(CS, ship)Cfcp, CusL)
21.                    ⇒ (if belong(CS, CusL)
22.          then getBill(CS, B) ask4Deliver(CS, ship) getCfIntfAsk4D(Cfcp, CusL)
23.                    else getCfIntfAsk4D(Cfcp, CusL) fi) .
```

[3] Again for simplicity, it was decided not to parameterize the shipment cost with arguments such as the weight, distance and time.

```
24. rl [getCfIntfAsk4Df ] : getCfIntfAsk4Df(Cfcpf getCfIntfAsk4D(Cfcp, CusL)) ⇒ Cfcpf .
25. endm
```

As mentioned already, the property *bill* is required from the history service interface. In the following formalisation of that service interface, it is extracted from the history service interface *HIS_INTF4SHIP_GNR* given in Appendix B, Figure B.24. That is, the rules getCfIntfSpd and getCfIntfSpdf intercept the bill.

```
1. mod HIS_INTF4SHIP_GNR is
. . .
7. op HIS : → Intf_NM .
8. op CusId: _: CustId → obs_Prop .
9. op bill:_ : Rat → obs_Prop .
. . .
18. rl [getCfIntfSpd]: getCfIntfSpd(< His | CusId: Cus > < His | bill: B > Cfcp, HisL)
19.                    ⇒ (if belong(His, HisL)
20.                    then < His | CusId: Cus > < His | bill: B > getCfIntfSpd(Cfcp, HisL)
21.                    else getCfIntfSpd(Cfcp, HisL) fi) .
22. rl [getCfIntfSpdf ] : getCfIntfSpdf(Cfcpf getCfIntfSpd(Cfcp, HisL)) ⇒ Cfcpf .
23. endm
```

5.6.2.2 The ECA-driven Shipment Rule Using the Extended Maude

The following depicts the Maude-based formalisation of the above ECA-based shipment rule. After the rule name SHIPMENT (line 9), the extra externalised delivery cost is given (line 10) (Appendix B, Figure B.25). As described in the ECA-driven interaction rule, the three shipment alternatives are defined and initialised as *TakeYselfPostage*, *NormalPostage* and *ExpressPostage* (lines 11-13). The interaction rule SHIPMENT, defined from lines 18 to 23, checks that the bill is greater than zero. A new bill is then computed by adding the postage cost and recorded in the history of that customer.

```
1. mod COORD_SHIP is
. . .
8. ops TakeYselfPostage NormalPostage ExpressPostage : → Postage
9. op SHIPMENT : → Coord_ NM .
10. op postage: _: Int → Attribute [ctor gather (&)] .
11. eq TakeYselfPostage = 0 .
12. eq NormalPostage = 20 .
13. eq ExpressPostage = 40 .
. . .
18. crl [SHIPMENT] : [SHIPMENT || (CS $ His $ Prvd $ ShipCom) @ postage: post]
19.                  & [CUST | ask4Deliver(CS, ship)] & [CUST | getBill(CS, B1)]
```

```
20.          & [His | CusId: CS] & [His | bill: B]
21.       ⇒  [SHIPMENT || (CS $ His $ Prvd $ ShipCom) @ postage: post]
22.          & [CUST | getBill(CS, (B1 + post))] & [His | CusId: CS] & [His | bill: B]
23.          if B > 0 .
24. endm
```

5.6.2.3 Dynamic Weaving of the ECA-driven Shipment Rule Using Extended Maude

The shipment process is controlled by the strategy *ASP_SHIP_Str*, as given in Appendix B, Figure B.26 and depicted in simplified form below. As detailed with respect to the other business activities above, interception, propagation, execution and weaving are required. Lines 15-26 intercept the events *cancelPr* and *getBill* from the customer service. This interception is governed by the rules getCfIntfAsk4D and getCfIntfAsk4Df. Similarly, *CusId* and *bill* are intercepted from the history service using the rules getCfIntfSpd and getCfIntfSpdf. What follows are the preparation and merging of these intercepted service interface states using the Subsume rule (line 28-30). After the execution of this interaction rule as an advice, the result is dispatched to different service interface states (line 39), then dynamically woven into running service components.

```
1. mod ASP_SHIP_Str is
. . .
10.op Compute : Term Nat → Term .
11.ceq Compute(T, N)
12.= if(N == 1) then
13.(if(SplitAT? :: Result4Tuple)
14.then Compute(getTerm(SplitAT?), N)
15.else if(belong? :: Result4Tuple)
16.then Compute(getTerm(belong?),N)
17.else if(getCfIntfAsk4D? :: Result4Tuple)
18.then Compute(getTerm(getCfIntfAsk4D?),N)
19.else if(getCfIntfAsk4Df?::Result4Tuple)
20.then Compute(getTerm(getCfIntfAsk4Df?),N)
21.else if(getCfIntfSpd? :: Result4Tuple)
22.then Compute(getTerm(getCfIntfSpd?),N)
23.else if(getCfIntfSpdf?::Result4Tuple)
24.then Compute(getTerm(getCfIntfSpdf?),N)
25.else if(intercept? :: Result4Tuple)
26.then Compute(getTerm(intercept?),N)
27.else if(Subsume? :: Result4Tuple)
28.then Compute(getTerm(Subsume?), N)
29.else if(Split_CfIntf? :: Result4Tuple)
30.then Compute(getTerm(Split_CfIntf?),N)
31.else if(SHIPMENT? :: Result4Tuple)
32.then Compute(getTerm(SHIPMENT?), N)
33.else Compute(T, 0)
. . .
35.else
36.(if(Recombin CfIntf?::Result4Tuple)
37.then Compute(getTerm(Recombin CfIntf?),0)
38.else if extractCfIntf?::Result4Tuple
39.then Compute(getTerm(extractCfIntf?),0)
40.else if (weaveCfIntf?::Result4Tuple)
41.then Compute(getTerm(weaveCfIntf?),0)
42.else if(RemoveNil? :: Result4Tuple)
43.then Compute(getTerm(RemoveNil?),0)
44.else if(RecombineAT?::Result4Tuple)
45.then Compute(getTerm(RecombineAT?),0)
. . .
```

5.7 The Payment Activity: The Approach at Work

The only payment alternative considered here, for the sake of simplicity, is using a bank card. Other possibilities such as credit card, bank transfer or cash payment can easily be captured as externalised ECA-driven business rules. The main idea of the rule addressed here is that following verification that the account to which the bank card applies has sufficient funds, the customer transfers the amount of the bill from it to the provider's account.

5.7.1 The Payment Activity (*Phases-rule+Int*): ECA-driven Architectural Modelling

A possible architectural modelling of this informal description of the payment rule is proposed in Figure 5.8. The services involved are the customer; his/her bank card and account. The customer has to trigger the payment using the event *ask4Pay*. This customer service must also supply the invoice to pay, that is, the resulting message *getBill* from the above activities. From the account service interface *AcntPaySI*, the transfer operation is required; it is proposed that the customer transfers the amount directly to the provider's account, assuming that the bank card is always in conformity with the associated account.

The ECA-driven payment rule is detailed on the left-hand side of Figure 5.8. The intercepted event *ask4Pay* represents the trigger of that rule. Under the constraint $bal \geq M$, the message transfer is performed. When it is successful, the customer receives a successful transfer event *succPay*.

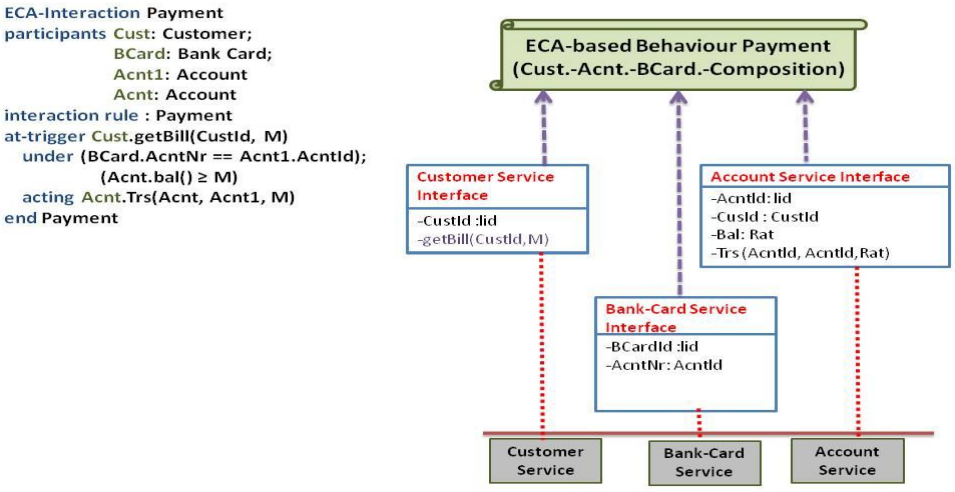


Figure 5.8: ECA-driven pattern of e-commerce for payment interactions

5.7.2 The Shipment Activity (*Phase-Maude*): Aspectual Maude at Work

5.7.2.1 Service Components and Interfaces: Formalisation and Validation in the Extended Maude

It is important to note that both the customer and the account service components have already been formalised in the extended Maude. The complete customer specification is given in Appendix B section B.1.5 and the account formalisation in Appendix A. The bank card service is trivial and skipped here, but is fully specified in Appendix B, Figure B.5.

As explained in respect of the ECA-driven architectural payment rule, the event *ask4Pay* is intercepted as a trigger in the interaction rule. The customer is informed of how much (s)he should pay through the event *getBill*. Therefore, in the following part of the Maude-based customer service interface module *CUS_INTF4PAY_GNR*,(fully specified in Appendix B, Figure B.27) both events *ask4Pay* and *getBill* are intercepted through the rules getCfIntfAsk4P and getCfIntfAsk4Pf (lines 19-23).

```
1. mod CUS_INTF4PAY_GNR is
• • •
8. op CUST : → Intf_NM .
9. op getBill( _, _ ) : CustId Rat → GB .
10. op ask4Pay( _ ) : CustId → Ask4P .
11. op succPay( _ ) : CustId → SuccP .
• • •
19. rl [getCfIntfAsk4P] : getCfIntfAsk4P(getBill(CS, B) ask4Pay(CS) Cfcp, CusL)
20.                 ⇒ (if belong(CS, CusL)
21.                 then getBill(CS, B) ask4Pay(CS) getCfIntfAsk4P(Cfcp, CusL)
22.                 else getCfIntfAsk4P(Cfcp, CusL) fi) .
23. rl [getCfIntfAsk4Pf ] : getCfIntfAsk4Pf(Cfcpf getCfIntfAsk4P(Cfcp, CusL)) ⇒ Cfcpf .
24. endm
```

In order to intercept the property *bal* from the account service component, getCfIntfbal and getCfIntfbalf and their rules are defined in lines 21-25. The rules *pendTRS* and *pendTRSf* are also defined, to pend the existing transfer messages *Trs* in the account component instance before the interception, while the rule *backTRS* reactivates the pending transfer messages to the corresponding account component instance. The corresponding complete specification, implemented in Maude Workstation, is depicted in Appendix B, Figure B.28.

```
1. mod ACNT_INTF4PAY_GNR is
• • •
8. op ACNT : → Intf_NM .
9. op Trs( _ , _ , _ ) : AcntId AcntId Rat → TRS .
10. op bal:_ : Rat → obs_Prop [ctor gather (&)] .
• • •
21. rl [getCfIntfbal] : getCfIntfbal(< AC | bal: B > Cfcp, AcntsL)
22.          ⇒ (if belong(AC, AcntsL)
23.          then < AC | bal: B > getCfIntfbal(Cfcp, AcntsL)
24.          else getCfIntfbal(Cfcp, AcntsL) fi) .
25. rl [getCfIntfbalf ] : getCfIntfbalf(Cfcpf getCfIntfbal(Cfcp, AcntsL)) ⇒ Cfcpf .
26. rl [pendTRS] : pendTRS(Trs(AC1, AC2, M) Cfcp)
27.          ⇒ Trs(AC1, AC2, M) pendTRS(Cfcp) .
28. rl [pendTRSf ] : pendTRSf(Cfcpf pendTRS(Cfcp)) ⇒ Cfcpf .
29. rl [backTRS] : backTRS(Cfcpf, Cfcp) ⇒ Cfcpf Cfcp .
30. endm
```

In the bank card interface *BCARD_INTF4PAY_GNR* below (Appendix B Figure B.29), the account number is intercepted through rules getCfIntfBC and getCfIntfBCf, defined from lines 15 to 19.

```
1. mod BCARD_INTF4PAY_GNR is
• • •
6. op BCARD : → Intf_NM .
7. op AcntNr: _: AcntId → obs_Prop [ctor gather (&)].
• • •
15. rl [getCfIntfBC] : getCfIntfBC(< BC | AcntNr: AC > Cfcp, BCardL)
16.          ⇒ (if belong(BC, BCardL)
17.          then < BC | AcntNr: AC > getCfIntfBC(Cfcp, BCardL)
18.          else getCfIntfBC(Cfcp, BCardL) fi) .
19. rl [getCfIntfBCf ] : getCfIntfBCf(Cfcpf getCfIntfBC(Cfcp, BCardL)) ⇒ Cfcpf .
20. endm
```

5.7.2.2 Formalisation of the ECA-driven Payment Rule Using Extended Maude

In this Maude-based formalisation, an extra attribute *acnt* is first declared to refer to an account for the provider. This is the account into which the customer has to transfer the bill amount. The specific Maude-based interaction rule is defined from lines 12 to 18. That is, under the constraint that the account suffices, the bill amount is transferred to the provider's account. In Appendix B, Figure B.30 the complete specification

```
1. mod COORD_PAY is
. . .
7. op Payment : → Coord_NM .
8. op acnt:_ : AcntId → Attribute [ctor gather (&)] .
. . .
12. crl [Payment] : [Payment || (CS $ BC $ AC) @ acnt: AC1] & [CUST | ask4Pay(CS)]
13.                 & [CUST | getBill(CS, M)] & [BCARD | < BC | AcntNr: AC >]
14.                 & [ACNT | < AC | bal: B >] & [ACNT | < AC1 | bal: B1 >]
15.              ⇒ [Payment || (CS $ BC $ AC) @ acnt: AC1] & [CUST | succPay(CS)]
16.              & [ACNT | Trs(AC, AC1, M) < AC | bal: B > < AC1 | bal: B1 >]
17.                 & [BCARD | < BC | AcntNr: AC >]
18.                 if B ≥ M .
19. endm
```

5.7.2.3 Dynamic Weaving of the ECA-driven Shipment Rule Using Extended Maude

The strategy for the payment process is explained using the following essential extract of the complete specification of the payment strategy detailed in Appendix B, Figure B.31. The first step in this payment strategy is to prepare for the interception by splitting the service component states through the *SplitAT* rule and pending the entire existing message *Trs* in the component instance, before proceeding to the payment activity, using the rules pendTRS and pendTRSf.

Next, lines 21 to 36 intercept all the required events and properties from the associated service component instances. From lines 23 to 26, the rules getCfIntfAsk4P and getCfIntfAsk4Pf are implemented to intercept the events *ask4Pay* and *getBill* from the customer. Lines 27 to 30 perform the rules getCfIntfbal and getCfIntfbalf, allowing the balance to be intercepted.

These intercepted service interface states (customer, account and bank card) are first prepared and merged with the ECA-driven interaction rule state (line 38). Once that rule is performed it is then dynamically woven into the corresponding service components, which are mainly the customer account (lines 49-52). Cleaning and recombining are then performed.

```
1.  mod ASP_PAY_Str is
      . . .
12.  op Compute : Term Nat → Term .
13.  ceq Compute(T, N)
14.        = if(N == 1) then
15.  (if(SplitAT? :: Result4Tuple)
35.  else if(intercept? :: Result4Tuple)
36.  then Compute(getTerm(intercept?),N)
37.  else if(Subsume? :: Result4Tuple)
38.  then Compute(getTerm(Subsume?), N)
39.  else if(Split CfIntf? :: Result4Tuple)
```

```
16. then Compute(getTerm(SplitAT?), N)
17. (if(pendTRS? :: Result4Tuple)
18. then Compute(getTerm(pendTRS?), N)
19. (if(pendTRSf? :: Result4Tuple)
20. then Compute(getTerm(pendTRSf?), N)
21. else if(belong? :: Result4Tuple)
22. then Compute(getTerm(belong?),N)
23. else if(getCfIntfAsk4P?::Result4Tuple)
24. then
      Compute(getTerm(getCfIntfAsk4P?),N)
25. else if(getCfIntfAsk4Pf?::Result4Tuple)
26. then
      Compute(getTerm(getCfIntfAsk4Pf?),N)
27. else if(getCfIntfbal? :: Result4Tuple)
28. then Compute(getTerm(getCfIntfbal?),N)
29. else if(getCfIntfbalf?::Result4Tuple)
30. then Compute(getTerm(getCfIntfbalf?),N)
31. else if(getCfIntfBC? :: Result4Tuple)
32. then Compute(getTerm(getCfIntfBC?),N)
33. else if(getCfIntfBCf? :: Result4Tuple)
34. then Compute(getTerm(getCfIntfBCf?),N)
```

```
40. then Compute(getTerm(Split CfIntf?),N)
41. else if(Payment? :: Result4Tuple)
42. then Compute(getTerm(Payment?), N)
      . . .
45. else
46. (if (Recombin_CfIntf? :: Result4Tuple)
47. then
      Compute(getTerm(Recombin_CfIntf?),0)
48. else if(extractCfIntf? :: Result4Tuple)
49. then Compute(getTerm(extractCfIntf?), 0)
50. else if (weaveCfIntf? :: Result4Tuple)
51. then (if(transfer? :: Result4Tuple)
52. then Compute(getTerm(transfer?),0)
53. else Compute(getTerm(weaveCfIntf?),0)
54. fi)
55. else if(backTRS? :: Result4Tuple)
56. then Compute(getTerm(backTRS?),0)
57. else if(RemoveNil? :: Result4Tuple)
58. then Compute(getTerm(RemoveNil?),0)
59. else if(RecombineAT? :: Result4Tuple)
60. then Compute(getTerm(RecombineAT?),0)
      . . .
```

5.8 The Change-Profile Activity: The Approach at Work

In the same spirit, towards externalising the business logic from any business activity, in terms of behavioural ECA-driven business rules, a demonstration is given in this section of how to achieve that externalisation even for unusual activities such as a change of profile. Indeed, the change of profile is usually hard-coded with the provider service component; thus it is not scaled up to an independent business activity. The severe limitations of this ordinary practice and decision are manifold. First, the rules governing profile change cannot be updated flexibly and dynamically, since their governing behaviour is hard-coded with the provider service. Second, when such rules increase in number, which are frequently the case, their accuracy and consistency become rapidly problematic. Third, putting the change of profile behaviour inside the provider means that cross-cutting code will be required from the customer and history, leading to hard and scattered pieces of code. Last but not least, the dynamic adaptability of such rules in a transparent and efficient manner becomes impossible, since they are not separated from the service component computations.

The aim of this section consists thus in following the same methodology, by considering the change of profile as an independent business activity of the e-shopping business process. Doing this will demonstrate how all the above shortcomings are circumvented. Moreover, there will be a free choice as to where to incorporate this business activity in the complete e-shopping business process. For instance, it can be incorporated after the confirmation or after

the payment activity, or it can even be performed independently and regularly (weekly, monthly, etc.). Indeed, the flexibility of a customer's profile depends on the frequency and amount of his/her spending. For instance, after a successful payment the spending changes. The spending can also be updated automatically within a regular period of time. Moreover, it can also be decreased by regular checking if it is not updated during a specific period of time. Consequently, there will be at least the two ways of updating a given profile: *changeProfile* and *RegularChangeProfile*. However, as indicated, there may be more sophisticated rules governing the profile update, which can now be directly described at the interaction level.

5.8.1 The Change-profile Activity (P*hases-rule+Int*): ECA-driven Architectural Modelling

The following proposal is thus for two variants of ECA-driven rules governing the updating of a profile: (1) after payment and (2) at regular periods. The first variant, as depicted in Figure 5.9, requires the participation of at least two service interfaces: the customer and his/her service history. From the customer service interface is required the triggering event, which may be *getBill*; that is, the rule is applied after each payment of the total bill. From the history service are required the bill and the date of the last update. At the interaction level, the properties are defined that set the qualifying limits for silver or gold status. For instance, with *MinSilver* set to £1000, if spending is under £1000 the customer's status is normal, while when it goes beyond *MaxSilver*, the customer is awarded a gold profile. Note that the attribute *currentDay* is also required in order to record the date of any such change.

```
ECA-Interaction Change Profile
participants Cust: Customer;
             His: History;
constants MinSilver = 1000;
          MaxSilver = 3000
attributes currentday : Date
interaction rule : Change Profile
at-trigger Cust.getBill(CustId, Rat)
      acting His.UpdateSp(Cust, bill);
             His.bill = 0;
             His.LastUpdateSp(currentDay)
      under1 (Cust.Profile == Normal)
             (His.Spending() ≥ MinSilver)
      acting1 Cust.Profile = Silver;
      under2 (Cust.Profile == Silver)
             (His.Spending() + His.bill() ≥ MaxSilver)
      acting2 Cust.Profile = Golden;
end Change Profile
```

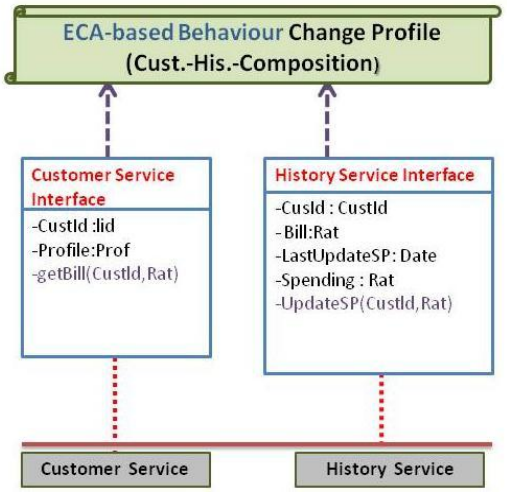


Figure 5.9: ECA-driven pattern of e-commerce for change profile interactions

For the regular update, as depicted in Figure 5.10, the interfaces required are again the customer and his/her history service. This activity is thus triggered automatically after each period of time (i.e. the attribute interval). Then, different formulas can be expressed. The event getCurrentDay intercepted from the history service interface will trigger the activity. If the spending of a customer is not updated during a specific period (for instance, if it remains unchanged after 3 months), it is reduced automatically by 100 until it reaches 0. The update of the spending may result in a change of the customer's profile status, according to the specific business rules. If the spending of a gold customer is between 3000 and 3100 and is not updated for more than 3 months, then his/her profile is changed to silver. Correspondingly, if the spending of a silver customer is between 1000 and 1100 and is not updated for more than 3 months, then his/her profile is changed to normal.

```
ECA-Interaction Regular Change Profile
participants Cust: Customer;
              His: History;
constants INTERVAL = 3
attributes interval : Int
interaction rule : Regular Change Profile
at-trigger1 His.getCurrentDay(HisId, Date)
  under1  (GreaterM(Date, His.LastUpdateSp,
                    interval) == true)
          (Cust.Profile == Golden)
          (His.Spending ≤ (MaxSilver + 100))
  acting1  Cust.Profile = Silver;
           His.LastUpdateSp = Date;
           His.Spending = (His.Spending - 100)
at-trigger2 His.getCurrentDay(HisId, Date)
  under2   (GreaterM(Date, His.LastUpdateSp,
                    interval) == true)
           (Cust.Profile == Silver)
           (His.Spending ≤ (MinSilver + 100))
  acting2   Cust.Profile = Normal;
            His.LastUpdateSp = Date;
            His.Spending = (His.Spending - 100)
at-trigger3 His.getCurrentDay(HisId, Date)
under3 (GreaterM(Date, His.LastUpdateSp,
                 interval) == true)
       (His.Spending ≥ 100)
acting3 His.LastUpdateSp = Date;
        His.Spending = (His.Spending - 100)
end Regular Change Profile
```

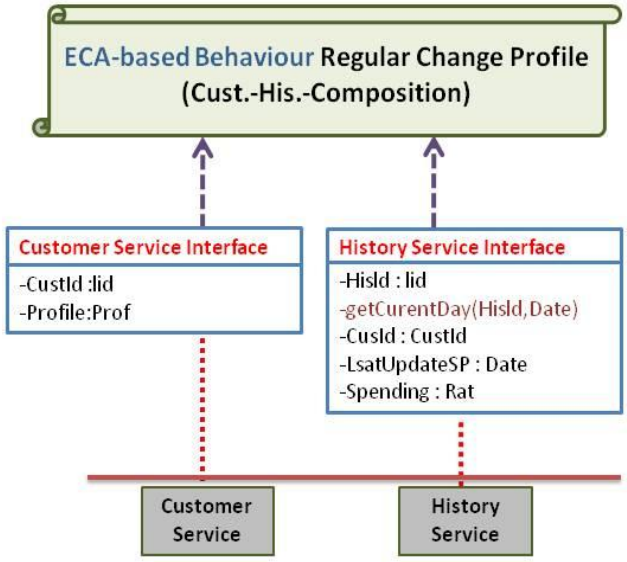


Figure 5.10: ECA-driven pattern of e-commerce for regular change-profile interactions

5.8.2 The Change-Profile Activity (*Phase-Maude*): Aspectual Maude at Work

5.8.2.1 Service Components and Interfaces: Formalisation and Validation in the Extended Maude

For this activity, the service interfaces required are the customer and his/her history. These have already been formalised (Appendix B, Figure B.1 and B.4). To achieve a Maude-based formalisation of these two ECA-driven architectural service interactions, the following (extract from the) customer service interface is first required. The profile property *Profile* and the event *succPay* are thus required from the customer. The event *succPay* is used only in the change-profile interaction rule and is intercepted through rules getCfIntfSuccP and getCfIntfSuccPf (lines 26-30). In Appendix B, Figure B.32 the complete specification.

```
1. mod CUS_INTF4CHGPF_GNR is
• • •
8. op CUST : → Intf_NM .
9. op succPay( _ ) : CustId → SuccP .
10. ops Normal Silver Golden : → Prof .
11. op Profile: _: Prof → obs_Prop .
• • •
21. rl [getCfIntfProf ] : getCfIntfProf(< CS | Profile: prf > Cfcp, CusL)
22.          ⇒ (if belong(CS, CusL)
23.          then < CS | Profile: prf > getCfIntfProf(Cfcp, CusL)
24.          else getCfIntfProf(Cfcp, CusL) fi) .
25. rl [getCfIntfProff] : getCfIntfProff(Cfcpf getCfIntfProf(Cfcp, CusL)) ⇒ Cfcpf .
26. rl [getCfIntfSuccP] : getCfIntfSuccP(succPay(CS) Cfcp, CusL)
27.          ⇒ (if belong(CS, CusL)
28.          then succPay(CS) getCfIntfSuccP(Cfcp, CusL)
29.          else getCfIntfSuccP(Cfcp, CusL) fi) .
30. rl [getCfIntfSuccPf ] : getCfIntfSuccPf(Cfcpf getCfIntfSuccP(Cfcp, CusL)) ⇒ Cfcpf .
31. endm
```

From the history service interface, the properties required are *spending*, *bill* and *LastUpdateSP*. The observed message *UpdateSP* and the triggering event *getCurrentDay* are also required. The rules getCfIntfSpd and getCfIntfSpdf are defined to intercept all these required properties. In Appendix B, Figure B.33 the complete specification.

```
1. mod HIS_INTF4CHGPF_GNR is
. . .
10. op HIS : → Intf_NM .
11. op CusId: _: CustId → obs_Prop .
12. op UpdateSp( _,_ ) : CustId Rat → UPDSP [ctor].
13. op bill:_ : Rat → obs_Prop .
14. op Spending:_ : Rat → obs_Prop .
15. op LastUpdateSP: _: Date → obs_Prop .
16. op getCurrentDay( _,_ ) : HisId Date → GCDAY [ctor].
. . .
26. rl [getCfIntfSpd]: getCfIntfSpd(getCurrentDay(His, today) < His | CusId: Cus >
27.                          < His | bill: B > < His | LastUpdateSP: date >
28.                          < His | Spending: M > Cfcp, HisL)
29.              ⇒ (if belong(His, HisL)
30.              then getCurrentDay(His, today) < His | CusId: Cus > < His | bill: B >
31.              < His | LastUpdateSP: date > < His | Spending: M > getCfIntfSpd(Cfcp, HisL)
32.              else getCfIntfSpd(Cfcp, HisL) fi) .
33. rl [getCfIntfSpdf ] : getCfIntfSpdf(Cfcpf getCfIntfSpd(Cfcp, HisL)) ⇒ Cfcpf .
34. endm
```

5.8.2.2 The Formalisation of ECA-driven Change-profile Rules Using Extended Maude

According to the description of the ECA-driven architectural interaction rules from Figures 5.9 and 5.10, the following Maude-based formalisation is proposed. The two rules CHGPF and RegulCHGPF capture the above ECA-driven architectural rules for the change of profile and the regular update of profile. In Appendix B, Figure B.34 the complete specification.

```
1. mod COORD_CHGPF is
. . .
6. op CHGPF : → Coord_NM .
7. op RegulCHGPF : → Coord_NM .
8. op interval:_ : Int → Attribute [ctor gather (&)] .
7.  op currentDay: _: Date → Attribute [ctor gather (&)] .
10. ops MinSilver MaxSilver INTERVAL : → Int .
11. eq MinSilver = 1000 .
12. eq MaxSilver = 3000 .
13. eq INTERVAL = 3 .
. . .
19. crl [CHGPF] : [CHGPF || (CS $ His) @ currentday: today] & [CUST | succPay(CS)]
20.                          & [CUST | < CS | Profile: prf >] & [HIS | < His | CusId: CS >]
21.                          & [HIS | < His | bill: B >] & [HIS | < His | LastUpdateSP: date >]
22.                           & [HIS | < His | Spending: S >]
23.              ⇒ if (prf == Normal and (S + B) > MinSilver)
24.  then [CHGPF ||(CS $ His) @ currentday: today]& [CUST | < CS |Profile: Silver >]
25.    & [HIS | UpdateSp(CS, B) < His | CusId: CS > < His | Spending: S >]
26.   & [HIS | < His | bill: 0 > < His | LastUpdateSP: today >]
27.   else if (prf == Silver and (S + B) > MaxSilver)
```

```
28.              then [CHGPF ||(CS $ His) @currentday: today]& [CUST |< CS |Profile: Golden >]
29.              & [HIS | UpdateSp(CS, B) < His | CusId: CS > < His | Spending: S >]
30.              & [HIS | < His | bill: 0 > < His | LastUpdateSP: today >]
31.  else if (B > 0)
32.              then [CHGPF || (CS $ His) @ currentday: today] & [CUST | < CS | Profile: prf >]
33.              & [HIS | UpdateSp(CS, B) < His | CusId: CS > < His | Spending: S >]
34.              & [HIS | < His | bill: 0 > < His | LastUpdateSP: today >]
35. else [CHGPF || (CS $ His) @ currentday: today] & [CUST | < CS | Profile: prf >]
36.               & [HIS | < His | bill: 0 >] & < His | CusId: CS >]
37.              & [HIS | < His | LastUpdateSP: date >] & [HIS | < His | Spending: S >] fi fi fi .
38.crl [RegulCHGPF] : [RegulCHGPF || (CS $ His) @ interval: Mt]
39.                        & [HIS | getCurrentDay(His, today)]
40.                        & [CUST | < CS | Profile: prf >] & [HIS | < His | CusId: CS >]
41.                        & [HIS | < His | LastUpdateSP: date >] & [HIS | < His | Spending: S >]
42.              ⇒  if(GreaterM(today, date, Mt) == true)
43. then (if((prf == Golden) and (S < 3100))
44.              then [RegulCHGPF ||(CS $ His)@interval: Mt] &[CUST |< CS|Profile: Silver >]
45.              & [HIS | < His | CusId: CS >] & [HIS | < His | LastUpdateSP: today >]
46.              & [HIS | < His | Spending: (S - 100) >]
47.else if((prf == Silver) and (S < 1100))
48.               then [RegulCHGPF ||(CS $ His)@ interval: Mt] & [CUST |< CS | Profile: Normal
                  >]
49.              & [HIS | < His | CusId: CS >] & [HIS | < His | LastUpdateSP: today >]
50.              & [HIS | < His | Spending: (S - 100) >]
51.else if(S > 100)
52.              then [RegulCHGPF || (CS $ His) @ interval: Mt] & [CUST | < CS | Profile: prf >]
53.              & [HIS | < His | CusId: CS >] & [HIS | < His | LastUpdateSP: today >]
54.              & [HIS | < His | Spending: (S - 100) >]
55.else [RegulCHGPF || (CS $ His) @ interval: Mt] & [CUST | < CS | Profile: prf >]
56.              & [HIS | < His | CusId: CS >] & [HIS | < His | LastUpdateSP: today >]
57.              & [HIS | < His | Spending: 0 >] fi fi fi)
58. else [RegulCHGPF || (CS $ His) @ interval: Mt] & [CUST | < CS | Profile: prf >]
59. & [HIS | < His | CusId: CS >] & [HIS | < His | LastUpdateSP: date >]
60. & [HIS | < His | Spending: S >] fi .
61. endm
```

5.8.2.3 Dynamic Weaving of ECA-driven Change-profile Rules Using Extended Maude

The ECA-driven interaction rules for the change of profile CHGPF and the regular change of profile RegulCHGPF are controlled by the strategy *ASP_CHGPF_Str*. This is fully specified in Appendix B, Figure B.35. As depicted in the above extracted part of that strategy, before the interception, all service interface states are to be split, through the *SplitAT* rule (line 15), then all required events and properties are intercepted from the customer and history (18-31). For instance, the successful-pay event *succPay* and the profile of the customer *Profile* are intercepted by rules getCfIntfSuccPf and getCfIntfProf. The rules getCfIntfSpd and getCfIntfSpdf allow the relevant events and properties to be

intercepted from the history interface. That is, it is necessary to intercept the rule getCurrentDay and the properties *CusId*, *bill*, *Spending* and *LastUpdateSP*.

These intercepted customer and history service interface states are first prepared and merged with the interaction state. At this stage, the interaction rules CHGPF and RegulCHGPF are performed (lines 36-39). Once these rules have been executed at the interaction level, it is next necessary to recombine any split parts of different resulting interfaces, using the rule Recombin_CfIntf in line 44. The different resulting service interface states are then separated using the rule extractCfIntf in line 46. Finally, they are woven into the respective service components using the rule weaveCfIntf (lines 47-51). The rules associated with different actions, such as the update-spending rule UpdateSp (line 49), are to be performed at the service component level. Lastly, any empty elements should be cleaned and all split elements recombined, using the rules RemoveNil and RecombineAT.

```
1. mod ASP_CHGPF_Str is
. . .
11.op Compute : Term Nat → Term .
12.ceq Compute(T, N)
13.= if(N == 1) then
14.(if(SplitAT? :: Result4Tuple)
15.then Compute(getTerm(SplitAT?), N)
16.else if(belong? :: Result4Tuple)
17.then Compute(getTerm(belong?),N)
18.else if(getCfIntfSuccP? ::Result4Tuple)
19.then Compute(getTerm(getCfIntfSuccP?),N)
20.else if(getCfIntfSuccPf?::Result4Tuple)
21.then Compute(getTerm(getCfIntfSuccPf?),N)
22.else if(getCfIntfProf? :: Result4Tuple)
23.then Compute(getTerm(getCfIntfProf?),N)
24.else if(getCfIntfProff?::Result4Tuple)
25.then Compute(getTerm(getCfIntfProff?),N)
26.else if(getCfIntfSpd? :: Result4Tuple)
27.then Compute(getTerm(getCfIntfSpd?),N)
28.else if(getCfIntfSpdf?::Result4Tuple)
29.then Compute(getTerm(getCfIntfSpdf?),N)
30.else if(intercept? :: Result4Tuple)
31.then Compute(getTerm(intercept?),N)
32.else if(Subsume? :: Result4Tuple)
33.then Compute(getTerm(Subsume?), N)
34.else if(Split_CfIntf? :: Result4Tuple)
35.then Compute(getTerm(Split_CfIntf?),N)
36.else if(CHGPF? :: Result4Tuple)
37.then Compute(getTerm(CHGPF?), N)
38.else if(RegulCHGPF? :: Result4Tuple)
39.then Compute(getTerm(RegulCHGPF?), N)
. . .
42.else
43.(if (Recombin_CfIntf? :: Result4Tuple)
44.then Compute(getTerm(Recombin_CfIntf?),0)
45.else if extractCfIntf? :: Result4Tuple
46.then Compute(getTerm(extractCfIntf?), 0)
47.else if (weaveCfIntf? :: Result4Tuple)
48.then (if(UpdateSp? :: Result4Tuple)
49. then Compute(getTerm(UpdateSp?),0)
50. else Compute(getTerm(weaveCfIntf?),0)fi)
51.else if(RemoveNil? :: Result4Tuple)
52.then Compute(getTerm(RemoveNil?),0)
53.else if(RecombineAT? :: Result4Tuple)
54.then Compute(getTerm(RecombineAT?),0)
```

5.9 Summary

This chapter has demonstrated how all of the phases of the integrated and progressive approach proposed in this thesis can be put into practice to develop runtime adaptive and knowledge-intensive service-oriented systems. More precisely, it has validated the practical

usability of the approach in an e-commerce application. The presentation of this case study has demonstrated the flexibility and simplicity of the application of the approach from different perspectives and at a number of levels. First, from the perspective of a separation of concerns, the approach can be applied with complete freedom either in breadth or in depth, or indeed in a combined way. That is, with respect to any business activity, it is possible to deal with all phases or to apply each phase to all business activities involved. More specifically, if the (intentional) rules for all business activities are available, the approach allows a simultaneous and parallel development of all activities to be achieved (using different designers). If they are not available, the designer can concentrate on any available activity rules and apply each step of the approach. From a reusability perspective, apart from the specificities of the rules, all service components are reused, as are the different patterns for progressing from one step to another. This accelerates greatly the prototyping of any complex application.

Chapter 6

Conclusions and Future Work

The main objective of the work reported in this thesis was to take a disciplined and progressive approach to behavioural interactions at the conceptual level, while developing dynamically adaptive service-oriented business applications. The thesis has proposed an integrated approach to address such inter-service behavioural interactions dynamically by covering early phases of business requirements and conceptualisation as well as their smooth and compliant validation using an adapted extension of Maude-based operational semantics. This concluding chapter first revisits some general considerations including the topic of research and its challenges, and then the main contributions made, finally offer some suggestions for further improvements and possible future work.

6.1 Thesis General Consideration

Harsh market competitiveness and globalisation have increasingly persuaded organisations to join their efforts and focus more on cooperation and interaction in order to add value to their businesses.

At the technological-level, this crucial shifting has been boosted by the emerging of new distributed architectures and programming paradigms of the underlying information systems of such cooperating organisational entities. The *service-oriented* paradigm represents nowadays the most advanced and widely-accepted technology, accurately implementing such loosely-coupled and networked large organisations.

Indeed, the service-oriented computing is endowed with a large range of standards facilitating the deployment and composition of different cross-organisational services. First, the concept of service is being universally defined using standardised interfaces, described using the XML-based standard WSDL. Such interfaces define all operations and messages to be exchanged with other services. Second, the exchanges and communication is also described using the XML-standard SOAP. Third, services can be universally (over the web) published using the respective standard language UDDI. Last but least, different services can

be discovered through their WSDL-based descriptions and then composed either in an orchestration-driven manner using BPEL4WS or in choreography-driven manner using WS-CDL.

Nevertheless, despite these technological advances underlying the service-oriented paradigm, several *crucial issues* are still hindering this technology to deliver all its promise. Among the serious shortcomings, this thesis have particularly been tackled in, the followings have been cited.

This work, thus aimed in at providing an integrated and stepwise solution for the development of reliable and dynamically adaptive service-oriented applications. The main objective was to ease and step-wisely support the development of complex service-oriented applications, while focussing on three main aspects: Explicitly deal with behavioural-intensive services, dynamically adapting composite services behavioural in a transparent and reusable manner and formally validating the development of such runtime dynamic behavioural compositional service applications.

This thesis explicit modelling of behavioural features of any elementary or composite services was based on the event-condition-actions ECA paradigm. Besides being intuitive, transparent yet abstract, this paradigm fits well with the event-driven nature of service-orientation and intrinsically supports the composition. The dynamic adaptation of such ECA-based service behaviour is in its turn based most recent advances in software-engineering, namely the aspect-oriented paradigm. Indeed, separate between the service description-level and its adaptability-level have been explicitly. Then advanced mechanisms have been proposed, for weaving and un-weaving any ECA-based service behaviour from the adaptability-level to the service-level in a stepwise and consistent manner. Also, it had been noted that at the adaptability-level, the behavioural ECA-rules can be manipulated, adapted and updated at need. Finally, the third main characteristic of this approach is the compliant formal setting, for a rigorous validation and rapid-prototyping. This objective has been achieved through a tailoring extension of rewriting-based Maude language and its reflection meta-level.

Through realistic case-studies dealing with Banking and E-Commerce applications, the methodology proposed in this thesis have been shown truly facilitates the development of dynamically adaptive and behavioural-intensive service-oriented applications. However, the

success and wide-adoption of any development approach cannot be achieved only on its own. That is to say, promising results from other complementary research initiatives, which fall outside the scope of this thesis, are also of paramount important and necessity, such as (i) *service requirements elicitation* including UML-based informal description of structural features of services and their composition; (ii) The *formal verification* of composite adaptive and behavioural features; and this in complementarities with the validation as it proposed in this work; (iii) The efficient and technology based implementation including a Java-based BPEL-based composition (iv) security and privacy issues are becoming more and more crucial preoccupations of service requestors and should thus be taken into account towards any accepted development approach.

6.2 Research Contribution

The main contributions of this thesis have been to:

- The *ECA-driven modelling of behavioural* service features and their composition, which includes extraction of behavioural issues and their elicitation and then their precise description as ECA rules. This also includes the explicit separation of such behavioural concerns from structural features of services.

- The *runtime adaptability* of (composite) services behaviour on the basis of these ECA-driven modelling service behaviours. This have been achieved through advanced techniques such as aspect-oriented techniques and reflection mechanisms

- The *compliant operational formal specification and validation* to ensure the correctness and increase the reliability of the approach. It does so through the tailoring of the rewriting-based object-oriented Maude language.

The *approach was assessment* through a non-trivial case study. This application deal with an E-shopping, where competition and customer preferences are at the centre of the offered services

6.2.1 ECA-driven Modelling of Behavioural Service Features

The service-oriented paradigm is event-driven by essence, as services are mainly invoked using triggering messages and operations. To remain compliant with and promote this advantage at the behavioural-level, to endow any service with richer behavioural features

expressed in terms of ECA rules have been proposed. More precisely, first any informal requirements of the service-oriented application at-hand have been assumed given. Such requirements could be mainly the objectives and goals to achieve through any alliance of specific organisations; the intentional rules governing the behaviour of such alliance; and business processes to be respected to achieve such goals. As a second step, then to refine any business activity in terms of its underlying event-driven ECA business rules has been proposed. For that purpose, a very appealing set of primitives to precisely describe such ECA-driven rules has been introduced. Last but least, in order to bridge the gap between this informal business-level and the service-oriented architectural-level, the described rules using architectural techniques have been leveraged, where all requirement features are expressed as service interfaces and the ECA-driven rules themselves are captured as connectors for composing such services.

6.2.2 Runtime Adaptability of Composite Service Behaviour

In the previous phase, it has been focussed on explicit and transparent description of service behaviour issues as well as their composition in terms of ECA-driven rules. Through this achievement, such rules explicitly could be adapt and manipulate, but only at design-time. To support runtime adaptability, which is essential as services are mostly discovered and invoked at runtime, it recapitulated from the aspect-oriented concepts and mechanisms. In this sense, a five-step methodology has been proposed for shifting-up the right invoked service-interfaces to the aspectual-level, where the ECA-driven rules are concerned. More precisely, first any events triggering interactions has been dynamically intercepted. Then, the right rules have been dynamically selected. Then any associated rule-centric service interactions has been perform. Afterwards, such emergent service behaviour on the running service components has been woven, in a non-intrusive manner.

6.2.3 Compliant Operational Specification and Validation

Although the two previous phases came up with precise and advanced mechanisms for dynamically dealing with runtime-adaptability in service-oriented applications, they remain at the descriptive-level. Furthermore, they could not directly execute at that abstract-level. To avoid any direct ad-hoc translation of these phases to the service technological-level and lose thereby most of the benefits, a compliant yet formal setting for validating such ECA-driven runtime adaptability of service behaviour have been proposed. It has been achieved

this through the leveraging of the rewriting-logic based Maude language and its reflection mechanisms. First how to endow the Maude concepts with the notion of service components and interfaces has been demonstrated, so that service-oriented applications as modelled in the previous phases can be formally specified and reasoned about. Then, the notion of ECA-driven rule-based service interactions as AOP advices has been integrated, by benefiting from the reflection capabilities of the Maude language. Finally, capitalising again from these reflection capabilities, how to dynamically weave the execution of such behavioural ECA-driven service interactions on participating service components via their service interfaces has been presented. For supporting the approach at the Maude-based implementation and validation, software that implements all the above described Maude extensions to fit our rule-centric aspectual architectural interactions have been developed, for adaptive service-oriented applications.

6.2.4 Approach Assessment with Case Studies

Along all the developed concepts of the forwarded approach, each issue using self-understandable banking example have been clarified and illustrated. This example served as a proof of concepts for all phases of that approach. In chapter five, a more promising non-trivial E-shopping case-study has been conducted. That is, all the three phases of the approach on that case-study has been projected. In this sense, and with respect to any service oriented business activity (e.g. order, confirm, payment, cancel, shipment), first its business logic as adapted event-driven business rules at the service interaction-level has been externalised. Then, these rules as ECA-driven architectural service connectors have been modelled. Finally, by using the compliant Maude developed tool they have been implemented and validated.

6.3 Future Work

In order to push forwards the presented approach in this thesis, at-least the following potential future research topics have been identified:

- The *Enhancement of the validation with formal verification,* which will complement the approach with further correctness and reliability.

- The *compliant deployment* using tailored service technology, which will promote a large adoption and acceptance of the approach outside the research circle.

- The *coping with advanced issues*, such as security and context-awareness, which will leverage the approach to multi-dimensional adaptive services.

- The *investment of other application domains*, including healthcare, energy, manufacturing. This will allow the approach to be become domain-specific and endow it with advanced modelling and adaptability patterns.

6.3.1. The Enhancement of the Validation with Formal Verification

The formal validation is an important step towards low-cost rapid-prototyping as well as detection of misconception errors, missing and misunderstanding between service customers and service providers. Consequently, it allows reshaping any conceptual modelling to fits the customer's requirements. Nevertheless, when it comes to the formal certification of the excepted properties to be fulfilled by the conceptual modelling, the validation on its own could not further help. Instead, verification techniques such as those based on temporal techniques should come into play. For that crucial aim, endowing the approach with such properties verification capabilities has been envisioned. A first straightforward direction in this sense would be the exploitation of the Maude built-in LTL (Linear-Temporal-Logic) model-cheeking tool for verification purpose. Further promising direction, would be to translate our ECA-driven architectural service interactions rules towards the ITL logic [93, 94]. As this thesis has been reported, this work is especially devoted to the verification of service behaviours and their composition. A first step towards such translation would be to enrich our ECA-driven rules with assumptions and commitments' assertions, and then benefits from the developed Tempura-ITL environment for doing verification. The translation of the checked properties to the developed ASDL language should also be of great benefits towards exposing such ECA-driven behaviour adaptive services.

6.3.2 The Compliant Deployment Using Tailored Service Technology

As this thesis has been emphasised, the forwarded approach has been focussing on the crucial early phases of informal, semi-formal, descriptive and finally formal setting, while developing adaptive service-oriented applications. That is, although the service-oriented paradigm at the architectural-level has been strictly followed, the mapping of the proposed conceptual solution using the available service technology was outside the thesis scope.

Therefore, it would be of paramount importance for the approach to be accepted at the technology-level, to address the compliant implementation of the approach using available and advanced service technology. Nevertheless, whereas the developed interfaces could be easily mapped to a standard WSDL-based description, the ECA-driven behavioural concerns as well as their runtime adaptive composition would represent challenging issues for available standards such as BPEL and WS-CDL. In other words, how to extend these standards should be addressed, with for instance aspect-oriented mechanisms, to cope with a compliant and correct translation to the service technology.

6.3.3 Coping with Advanced Issues: Security and Context Awareness

Within the era of internet, security concerns (e.g. access, privacy, trust) represent nowadays the must-be milestones in any solution around service technology. Being aware of such crucial importance of security issues, it envisions addressing them in any future investigations. In this sense and the stay compatible with the approach, first to capture any security constraints as ECA-driven rules and describe them separately at the adaptive aspectual-level. In this manner has been suggested, it can be dynamically handle them and impose the right ones at-need.

Along with security issues emerge also context-aware concerns, which are indeed nowadays intrinsically interrelated. While addressing dynamic adaptability in this thesis, service functionalities and interactions has been mainly concentrated. Nevertheless, context-aware issues such as service customer preferences, location as well as service conditions, should also be tackled at runtime. On the other hand, the security should be tailored to the requested services and their requestors, for an optimal control and personalisation of access and privacy patterns. All these advanced and very required features should be investigated and accordingly incorporated in the proposed approach to ensure its acceptance and adoption in the future.

6.3.4 Investment of other Application Domains

A successful approach for developing adaptive service-oriented applications should be a general-purpose and not specific to restricted application-domains. In this thesis, how the approach can be smoothly applied to applications from E-Commerce and E-Banking has been demonstrated. Elaborating on these experiences, it should be in the future extending the

application of the approach to other potential domains, such as healthcare, manufacturing, energy and E-Learning, just to name few ones. For instance, the healthcare domain is well-known for its agility, where doctors, medication, patients and hospital infrastructure have to be part of the targeted service-oriented applications. Any of these services present complex and challenging behavioural issues, besides intensive and event-driven interactions.

Such cross-application domains experiences should also lead us to the development of measured design patterns for each domain. Furthermore, we envision also that rules themselves and their patterns will be heavily influenced by the specificities of each domain.

References

[1] J.Abreu, L.Bocchi, J.L.Fiadeiro and A.Lopes, "Specifying and Composing Interaction Protocols for Service-Oriented System Modelling", *Formal Techniques for Networked and Distributed Systems (FORTE 2007),* LNCS 4574, Springer, pp. 358-373, 2007.

[2] R. Allen and G. Garlan. "Formalizing Architectural Connection. " *In Proc. Of the Sixteenth International Conference on Software Engineering,* pp 71–80, 1994.

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: concepts, architectures and applications.* Springer-Verlag, 2004.

[4] A. Alves e lt. "Web service business process execution language version 2.0." *Technical report, OASIS,* April, 2007.

[5] L. Andrade and J. Fiadeiro."Architecture Based Evolution of Software Systems. " *Lecture Notes in Computer Science,* vol 2804, pp 148–181. Springer, 2003.

[6] L. Andrade, J. Fiadeiro, J. Gouveia1, and G. Koutsoukos. "Separating computation, coordination and configuration". *Journal of Software Maintenance and Evolution: Research and Practice* 14, 353–369. 2002.

[7] T. Andrews et al. "Business Process Execution Language for Web Services (BPEL4WS) ". *V 1.1. IBM developer Works;*

[8] A. Ankolekar and et al. "DAML-S: Web Service Description for the Semantic Web. " *In Proc. of International Semantic Web Conference (ISWC), IEEE CS,* pp 348–363. 2002.

[9] A. Arsanjani. *Grammar-Oriented Object Design: Towards Dynamically Reconfigurable Business and Software Architecture for On-demand Computing*. Journal, PhD Thesis, publisher, De Montfort University.2003.

[10] A. Arsanjani.(2004). *Service-Oriented Modeling and Architecture (SOMA),* IBM Developer-Works,[Online].Available: http://www.ibm.com/developerworks/webservices/library/ws-soa-design1 .

[11] M. Bajec and M. Krisper. "A Methodology and Tool Support for Managing Business Rules in Organisations. Information Systems", 30(6):423–443, 2004.

[12] C. Baral and M. Gelfond. "Logic Programming and Knowledge Representation". *Journal of Logic Programming* 19, 20:73–148, 1994.

[13] A. Barros, M. Dumas, and P. Oaks. "A Critical Overview of the Web Services Choreography Description Language (WS-CDL)". *BPTrends,* 2005.

[14] T. Batista, C. Chavez, A. Garcia, A. Sant'Anna, U. Kulesza, A. Rashid, and F. Filho. "Reflections on Architectural Connection: Seven Issues on Aspects and ADLs." *In Proc. of Workshop on Early Aspects at ICSE'06*, 2006.

[15] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. 2003.

[16] M. Beisiegel, H. Blohm, D. Booz, and et al. "Service Component Architecture. Building Systems using a Service Oriented Architecture." *Technical report*, 2005.

[17] T. Berners-Lee, J. Hendler, and O. Lassila. "The Semantic Web." *Scientific American*, 284(5):34–43, 2001.

[18] David Booth. (February,2004). *Web Services Architecture*, W3C Working Group Note, [Online].Available: http://www.w3.org/TR/ws-arch/.

[19] Robert Brunner.(September,2002). *Web services and Flows (WSFL).* Sams Publishing, [Online].Available: www.developer.com/java/web/article.php/1462301.

[20] F.Casati, D.Georgakopoulos and M.C.Shan, "Technologies for E-Services", *Proc. of 2nd International Workshop on Technologies for E-Services*, pp. 16-29, LNCS, vol 2193, 2001.

[21] A. Charfi, R. Khalaf, and N. Mukhi. "QoS-Aware Web Service Compositions Using Non-intrusive Policy Attachment to BPEL". *In Proc. of 5th International Conference on Service Oriented Computing (ICSOC'07), Industry track, to appear. Springer,* September 2007.

[22] A. Charfi, M. Mezini "Hybrid Web service composition: business processes meet business rules", *Int'l Conf. on Service Oriented Computing (ICSOC 2004), New York*, Dec 2004.

[23] A. Charfi and M. Mezini. "AO4BPEL: An Aspect-oriented Extension to BPEL". *World Wide Web, vol10, nr. 3*, pp. 309-344, 2007.

[24] S. Cheng and D. Garlan. "Mapping architectural concepts to UML-RT". *In Proc. Of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001),* 2001.

[25] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI, 2002.

[26] M. Clavel, F. Duran, S. Eker, P. Lincoln, J.Meseguer, and J. Quesada. "Maude: specification and programming in rewriting logic". *Theoretical Computer Science*, vol 285, pp. 187–243, 2002.

[27] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C.L: Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science (Springer), 4350, 2007.

[28] M. Clavel and J. Meseguer. "Reflection and Strategies in rewriting logic". *In G. Kiczales, editor, Proc. of Reflection'96, Xerox PARC,* pp 263–288. 1996.

[29] M. Clavel and J. Meseguer. "Reflection in conditional rewriting logic". *Theoretical Computer Science*, vol 285, pp 245–288, 2002.

[30] Hewlett-Packard Company.(March,2002). *Web services Conversation Language (WSCL) 1.0.* W3C recommendation, [Online].Available: http://www.w3.org/TR/wscl10/.

[31] Cong-Vinh and Bowen. " Formal Approach to Aspect-Oriented Modular Reconfigurable Computing". *In First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07),* pp 369–378, 2007.

[32] D.Craigen, S.Gerhart, and T.Ralston, "Formal Methods Reality Check: Industrial usage", *IEEE Trans. Softw. Eng.* vol 21(2), pp. 90-98, 1995.

[33] C. Courbis and A. Finkelstein. "Towards aspect weaving applications". *In Proceedings of the 27th international conference on Software engineering (ICSE '05),* pp 69–77. ACM Press, 2005.

[34] G. Denker. "From Rewrite Theories to Temporal Logic Theories". *In H. Kirchner and C. Kirchner, editors, Proc. of Second International Workshop on Rewriting Logic, Electronic Notes in Theoretical Computer Science*, vol 15, 1998.

[35] G. Denkera, C. Talcotta, G. Rosu, S. Ekera, and T. Florin. "Rewriting Logic Systems". *Electronic Notes in Theoretical Computer Science*, vol 176, pp. 233–247, 2007.

[36] N.Dershowitz and J.Jouannaud. "Rewrite Systems". *In Handbook of Theoretical Computer Science, vol B: Formal Methods and Semantics*, chapter 6, pp. 243-320, 1990.

[37] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications 1 : Equations and initial semantics*. EATCS Monographs on Theoretical Computer Science, 1985.

[38] S. Eker, J. Meseguer, and A. Sridharanarayanan. "The Maude LTL Model Checker". *Electronic Notes in Theoretical Computer Science*, vol 71,pp.19–21, 2002.

[39] A. Erradi and P. Maheshwari. "Adaptive BPEL: Policy-Driven Middleware for Flexible Web Services Composition". *In Proc. of the EDOC 2005 Middleware for Web Services Workshop (MWS'05). IEEE Computer Society,* 2005.

[40] A. Erradi and P. Maheshwari, V. Tosic. "Policy-Driven Middleware for Self-adaptation of Web Services Compositions". *In ACM/IFIP/USENIX 7th International Middleware Conference, Lecture Notes in Computer Science*, vol 4290, pp. 62–80. Springer, 2006.

[41] A. Erradi, V. Tosic, and P. Maheshwari. "MASC -.NET-Based Middleware for Adaptive Composite Web Services". *In IEEE International Conference on Web Services (ICWS07), IEEE Computer Society)*, pp. 727–734. 2007.

[42] L.M. Eshkevari, V. Arnaoudova, and C. Constantinides. "Comprehension and Dependency Analysis of Aspect-Oriented Programs through Declarative Reasoning". *In Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008, Lecture Notes in Computer Science,* vol 4902, pp. 35–52, 2008.

[43] R. Filman, T. Elrad, S. Clarke, M. Aksit . *Aspect-Oriented Software Development*, 1st ed, Addison-Wesley Professional, 2004.

[44] Y. Fu, Z. Dong, and H. He. "An Approach to Web Services Oriented Modeling and Validation". *In of the 28th ICSE workshop on Service Oriented Software Engineering (SOSE2006). ACM Press*, 2006.

[45] L. Fuentes and P. S´anchez. "Towards executable aspect-oriented UML models". *In Proceedings of the 10th international workshop on Aspect-oriented modeling, ACM Press*, (AOM'07), pp 28–34. 2007.

[46] A. Garcia, C. Chavez, V. Batista, C. SantAnna, U. Kulesza, R. Awais, and C. Pereira de Lucena. "On the Modular Representation of Architectural Aspects". *In Third European Workshop on Software Architecture, EWSA 2006, Lecture Notes in Computer Science, Springer*, vol 4344, pp 82–97, 2006.

[47] J.A. Goguen and J. Meseguer. "Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations". *Theoretical Computer Science*, vol 105 ,pp 217–273, 1992.

[48] J. Goguen, T. Winkler, J. Meseguer, K. Gutatsugi, and J. P. Jouannaud. Introducing OBJ. In A. Goguen and G. Malcolm, editors, "Software Engineering with OBJ: Algebraic Specification in Action", *Kluwer Academic* , pp 3–167. 2000.

[49] J.H. Hausmann, R. Heckel, and M. Lohmann. "Model-based development of web service descriptions: Enabling a precise matching concept". *International Journal of Web Services Research*, 2(2):67–84, 2005.

[50] D. Hay and K. Healy. "Defining business rules ~ what are they really?" *Technical Report Revision 1.3, Business Rules Group,* 2000.

[51] K. Jensen. Coloured Petri Nets. *Basic concepts, analysis methods and practical use*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1996.

[52] J.Y. Jung, J. Park, S. Han, and L. Lee. "An ECA-based framework for decentralized coordination of ubiquitous web services". *Information Software Technology,* 49(11-12):1141–1161, 2007.

[53] P. Kardasis and P. Loucopoulos. "Expressing and organising business rules". *Information and Software Technology*, vol 46 , pp 701–718, 2004.

[54] P. Kardasis and P. Loucopoulos. "A Roadmap for the Elicitation of Business Rules in Information Systems Projects". *Business Process Management Journal*,11(4), pp. 316–348, 2005.

[55] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon and C. Barreto.(2005). *Web Services Choreography Description Language (WS-CDL) 1.0*. [Online].Available: http://www.w3.org/TR/ws-cdl-10/.

[56] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, and J. Palm. "An Overview of AspectJ". *Lecture Notes in Computer Science, Springer*. pp 327–353. 2001.

[57] G. et al. Kiczales. "Aspect-Oriented Programming". *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP).* Springer-Verlag LNCS 1241. June, 1997.

[58] J. Kramer. "Exoskeletal software". *In Proceedings 16th International Conference on Software Engineering. IEEE Computer Society*, 1994.

[59] H. Kreger. "Web services conceptual architecture (WSCA 1.0) ". *IBM Software Group*, May 2001.

[60] R. Laeammel. "Declarative aspect-oriented programming". *In Proceedings PEPM'99, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM'99, BRICS Notes Series NS-99-1,* pp 131–146, 1999.

[61] F. Leymann. "Web Services Flow Language (WSFL 1.0) ", *IBM Software Group,* May 2001.

[62] N. Lohmann, P Massuthe, C. Stahl, and D.Weinberg. "Analyzing interacting WSBPEL processes using flexible model generation". *Data & Knowledge Engineering*, vol 64(1), pp. 38-54, 2008.

[63] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. "Specifying Distributed Software Architectures". *In Proc. of the Fifth European Software Engineering Conference*, pp. 137-153,1995.

[64] J. Magee and J. Kramer. "Dynamic Structure in Software Architectures". *In 4th Symp. On Foundations of Software Engineering, ACM Press*, pp 3–14. 1996.

[65] J. Martin and J. Odell. *Object-Oriented Methods: a Foundation—UML Edition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[66] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. R. Payne, M. Sabou, M. Solanki, N. Srinivasan, K. Sycara (SRI, CMU, Univ. Toronto) "Bringing Semantics to Web Services: The OWL-S Approach." *First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), San Diego, California, USA*, July 6-9, 2004,

[67] N. Marti-Oliet and J. Meseguer. "Rewriting logic as a logical and semantic framework". *In J. Meseguer, editor, Proc. of First International Workshop on Rewriting Logic, Electronic Notes in Theoretical Computer Science*, vol 4, pp 189–224, 1996.

[68] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. "Using object-oriented typing to support architectural design in the C2 style". *In Proc. of the 4th ACM Symposium on the Foundations of Software Engineering*, 1996.

[69] N. Medvidovic and R. Taylor. "A framework for classifying and comparing architecture description languages". *In Proc. of Conceptual Modeling for Novel Application Domains, Lecture Notes in Computer Science. Springer*, vol 1303, 1997.

[70] N. Medvidovic and R.N. Taylor. "A Classification and Comparison Framework for Software Architecture Description Languages". *IEEE Transactions on Software Engineering,* 26(1):70–93, 2000.

[71] J. Meseguer. "Conditional rewriting logic as a unified model for concurrency". *Theoretical Computer Science*, Vol 96 , pp 73–155, 1992.

[72] J. Meseguer. "Rewriting logic as a semantic framework for concurrency". *A progress report. In U. Montanari and V. Sassone, editors, Proc. 7th Int. Conf. on CONCUR96: Concurrency Theory, Lecture Notes in Computer Science,* vol 1119, pp 331–372. Springer, 1996.

[73] J. Meseguer. "Membership algebra as a semantic framework for equational specification". *In F. Parisi-Presicce, editor, Proc. WADT'97, Lecture Notes in Computer Science,* vol 1376 , pp 18–61. Springer, 1998.

[74] J. Meseguer. "Research Directions in Rewriting Logic". *In U. Berger and H. Schwichtenberg, editors, Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany. Springer*, 1998.

[75] M. Mezini and K. Ostermann. "Conquering aspects with Caesar". *In Proc. of the 2nd international conference on Aspect-oriented software development*, pp 90–99. ACM, 2003.

[76] C. Montangero, L. Semini. "A logical view of choreography". *In Paolo Ciancarini and Herbert Wiklicky, editors, COORDINATION, Lecture Notes in Computer Science,* vol 4038 , pp 179–193. Springer, 2006

[77] H. Motahari, B. Benatallah, A. Martens, F. Curbera, and F. Casati. "Semi-Automated Adaptation of Service Interactions". *In The 16th International World Wide Web Conference (WWW2007)* pp 993–1002, 2007.

[78] C. Nagl, F. Rosenberg, and S. Dustdar. "ViDRE - A Distributed Service-Oriented Business Rule Engine based on RuleML". *In 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06). IEEE Computer Society*, 2006.

[79] S.Narayanan and S.McIlraith, "Analysis and Simulation of Web Services", *Computer Networks,* Vol 42, pp. 675-693, 2003.

[80] OASIS.(17 Apr, 2006): *UDDI Specification*, [Online].Available: http://uddi.xml.org/specification, Copyright (c) OASIS.

[81] B. Orri¨ens, J. Yang, and M.P. Papazoglou. "A Framework for Business Rule Driven Web Service Composition". *In Proc. of Conceptual Modeling for Novel Application Domains, Lecture Notes in Computer Science,* vol 2814 , pp 52–64. Springer, 2003.

[82] C. Ouyang, E. Verbeek, W.M.P. v. d. Aalst, S. Breutel, M. Dumas, A.H.M. t. Hofstede, "Formal Semantics and Analysis of Control Flow in WS-BPEL", *Technical report (revised version), Queensland University of Technology,* October 2005.

[83] M.P. Papazoglou. *Web Service: Principles and Technology*. Prentice-Hall, Englewood Cliffs, 2007.

[84] D. Perry and A. Wolf. "Foundations for the Study of Software Architectures". Vol 17, pp 40–52, 1992.

[85] W. Reisig. "Petri Nets and abstract data types". *Theoretical Computer Science*, vol 80, pp.1–30, 1991.

[86] Taylor R.N. "A Component- and Message-Based Architectural Style for GUI Software". *IEEE Transactions on Software Engineering, Vol. 22 No. 6, 390-406.* June, 1996.

[87] F. Rosenberg and S Dustdar. "Towards a Distributed Service-Oriented Business Rules System". *In Proc. of the of EEE European Conference on Web services (ECOWS). IEEE Computer Society Press*, 2005.

[88] RuleML.(2005). *The Rule Markup Initiative*. [Online].Available:www.ruleml.org.

[89] S. Ruy, B. Benatallah, and F. Casati. " Framework for Managing the Evolution of Business Protocols in Web Services". *In Asia-Pacific Conference on Conceptual Modelling (APCCM'07),* 2007.

[90] M. Shaw, R. DeLine, D. Klein, T. Ross, D. M. Young, and G. Zelesnik. "Abstractions for Software Architecture and Tools to Support Them". *IEEE Transactions on Software Engineering*, pp. 314–335, 1995.

[91] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1996.

[92] M. Snoeck and G. Poels. "Analogical Reuse of Structural and Behavioral Aspects of Event-Based Object-Oriented Domain Models". *International Workshop on Enterprise and Domain Engineering – DomE 2000 (within the DEXA 2000 Conference),* 2000.

[93] M. Solanki, A. Cau, and H. Zedan. "Introducing Compositionality in Web Service Descriptions". In *Proceedings of the International Conference on World Wide Web. IEEE Computer Society Press*, 2004.

[94] M.K.Solanki. *A Compositional Framework for the Specification, Verification and Runtime Validation of Reactive Web Services*. journal, PhD Thesis, publisher, De Montfort University.2005.

[95] S. Staab, W. van der Aalst, V. R. Benjamins, A. Sheth, J. A. Miller, C. Bussler, A. Maedche, D. Fensel, and D. Gannon. "Web Services: Been There, Done That?" *IEEE Intelligent Systems,* 18(1):72–85, Jan/Feb 2003.

[96] S. Ross-Talbot. "Orchestration and Choreography: Standards, Tools and Technologies for Distributed Workflows". *Workshop on Network Tools and Applications in Biology (NETTAB 2005), Naples, Italy*, October 2005.

[97] S.Thatte. *Xlang web services for business process design*, 2001. [Online].Available: http://www.gotdotnet.com/team/xml_wsspecs/xlang-c.

[98] Rapide Design Team. " DRAFT Rapide 1.0 Language Reference Manual". Program Analysis and Verification Group, Computer Systems Lab, Standford University, May 1996.

[99] Tsai, W.T., Song, W., Paul, R., Cao, Z. and Huang, H. "Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing". *In Proc. of the IEEE COMPSAC 2004*. pp 554–559, 2004.

[100] W.M.P. Van der Aslst, A. Barros, H.M. Hofstede, and B. Kiepuszewski. "Advanced workflow patterns". *In Proc. of COOPIS'2000,* pp 18–29, 2000.

[101] S. Vestal. "A cursory overview and comparison of four architecture description languages". *Technical report*, 1993.

[102] Y. Wang, S. Singh, J. Hosking, and J. Grundy. " An aspect-oriented UML tool for software development with early aspects". *In Proceedings of the 2006 international workshop on Early aspects at ICSE*, pp 51–58. ACM Press, 2006.

[103] Web-Services.(2001). *Web-Services Architect, Part 2: Models for dynamic-business,*[Online].Available: http://www-106.ibm.com/developerworks/webservices/library/ws-arc2.html.

[104] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and DF Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WSReliable Messaging, and More*. Prentice-Hall, Englewood cliffs, 2005.

[105] M. Wermelinger, A. Lopes, and J. Fiadeiro. "Superposing connectors". *In Proc. 10h International Workshop on Software Specification and Design, IEEE Computer Society Press 2000*, pp 87–94. 2000.

[106] W.M.N. Wan-Kadir and P. Loucopoulos. "Relating Evolving Business Rules to Software Design". *Journal of Systems Architecture*, 2003.

[107] Y. Yang, Q. Tan, and Y. Xiao. "Verifying Web Services Composition Based on Hierarchical Colored Petri Nets". *In Proceedings of IHIS05*, pp 47–53. ACM Press, 2005.

[108] X. Yi and K.J. Kochut. "A CP-nets-based Design and Verification Framework for Web Services Composition". *In Proceedings of 2004 IEEE International Conference on Web Services, IEEE Computer Society*, pp 756–760, 2004.

Bibliography

[1] W. M. P. V. Aalst. "Don‘t go with the flow: Web services composition standards exposed. " *IEEE Intelligent Systems*, 18:72–76, 2003.

[2] F. Assche, P. Layzell, and G. Speltincx P. Loucopoulos. "Information systems development: a rule-based approach." *Knowledge-Based Systems,* 1(4):227-234,19 1988.

[3] W. Beer, V. Christian, A. Ferscha, and L.Mehrmann. "Context-aware Behavior by Interpreted ECA Rules. " *In Proceedings of EUro-Par’03, Lecture Notes in Computer Science,* vol 2790 , pp 1064–1073, 2003.
www-06.ibm.com/developerworks/webservices/library/ws-bpel ,May 2003.

[4] N.Busi, R.Gorrieri, C.Guidi, R.Lucchi and G.Zavattaro, G., "Choreography and Orchestration: A Synergic Approach for System Design", *Proc. of 3rd International Conference on Service Oriented Computing (ICSOC'05),* pp.228-240, LNCS Vol 3826, Springer, 2005.

[5] C. Chvez and C. Lucena. "Design-level Support for Aspect Oriented Software Development. OOPSLA Workshop on Advanced Separation of Concerns", *Minneapolis, Minnesota*, October 2001.

[6] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. "Separating Concerns throughout the Development Lifecycle". *3rd AOP Workshop at ECOOP*, 1999.

[7] M. Clavel, F. Duran, S. Eker, J.Meseguer, and M.Stehr. "Maude: Specification and programming in rewriting logic." *Technical report,* (1999). http://maude.csl.sri.com.

[8] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. "Principles of Maude". *In Jos´e Meseguer, editor, Proceedings, First International Workshop on Rewriting Logic and its Applications. Elsevier Science. Electronic Notes in Theoretical Computer Science,* Vol 4,1996.

[9] L. Coglianese and R. Szymanski. "DSSA-ADAGE: An Environment for Architecture-based Avionics Development". *In Proc. of AGARD’93*, 1993.

[10] G. Diaz, M. Cambronero, J. Pardo, V. Valero and F. Cuartero. "Automatic generation of Correct Web Services Choreographies and Orchestrations with Model Checking Techniques". *In Telecommunications*, 2006.

[11] R. Douence, P. Fradet, and M. Sudholt. "Composition, reuse and interaction analysis of stateful aspects". *In Proc. In 4th Int. Conf. on Aspect-oriented Software Development (AOSD’04) ACM Press*, pp 141–150. 2004.

[12] R. Farahbod, U. Glaesser, and M. Vajihollahi. "Specification and Validation of the Business Process Execution Language for Web Services". *In W. Zimmermann and B. Thalheim, editors, Proc. of ASM’2004, Lecture Notes in Computer Science ,Springer.* pp 78–94, vol 3052, 2004.

[13] D. Garcia and M. de Toledo. "Semantics-enriched QoS policies for web service interactions". *In Proceedings of the 12th Brazilian symposium on Multimedia and the web (WebMedia'06), ACM Press,* pp 35–44. 2006.

[14] D. Garlan, A. Kompanek, R. Melton, and R. Monroe. "Architectural Style: An Object-Oriented Approach". 1996.

[15] D. Garlan, R. Monroe, and D. Wile. "Acme: An Architecture Description Interchange Language". *In Proc. of CASCON'97*, pp 169–183, 1997.

[16] J. Goguen and R. Diaconescu. "An Oxford Survey of Order Sorted Algebra". *Technical report*, 1994.

[17] M. Gudgin, M. Hadley, J. Moreau and H. Nielsen . "SOAP Version 1.2 Part 1: Messaging Framework". W3C *Recommendation, World Wide Web Consortium*, June 2003.

[18] R. Heckel, J. Kuester, S. Thene, and H. Voigt. "Towards Consistency of Web Service Architectures". *In 7th World Multiconference on Systemics, Cybernetics, and Informatics (SCI 2003),* 2003.

[19] C. Hofmeister, R. Nord, and D. Soni. "Describing software architecture with UML". *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), pp. 145-159, San Antonio, TX*, February 1999.

[20] D.Hunter, editor. *Beginning XML*. Wiley Publishing, 2004.

[21] I.J. Jureta, S. Faulkner, and P. Thiran. "Dynamic Requirements Specification for Adaptable and Open Service-Oriented Systems". *In Proc. of 5nd International Conference on Service Oriented Computing (ICSOC'07), Lecture Notes in Computer Science*, vol 4749 ,pp 270–282. Springer, 2007.

[22] T. Berners-Lee, J.Hendler and Ora Lassila. "The Semantic Web". *Scientific Amrican,* May 2001.

[23] Z. Maamar, G. AlKhatib, S. Most´efaoui, M. Lahkim, and W. Mansoor. "Context-based Personalization of Web Services Composition and Provisioning". *In Proc. EUROMICRO'04. IEEE Computer Society,* 2004.

[24] A. Martens. "Analysing Web Service Based Business Processes". *In M.Cerioli, editor, Proceedings of the Fundamental Approaches to Software-Engineering (FASE'05),* pp 19–33. LNCS 3442, 2005.

[25] N. Medvidovic and D. Rosenblum. "Assessing the suitability of a standard design method for modeling software architectures". *In Proc. of the First Working IFIP Conference on Software Architecture (WICSA1)*, 1999.

[26] J. Meseguer. "Rewriting Logic as a unified Model of Concurrency". *In J. C. M. and J. W. Klop, editors, Proc. of CONCUR'90, Lecture Notes in Computer Science*, vol 458, pp 384–400, 1990.

[27] J. Meseguer. "A Logical Theory of Concurrent Objects and its Realisation in the Maude Language". *In P. Wegner G. Agha and A. Yonezawa, editors, Research Directions in Object-Based Concurrency*, pp 314– 390, 1993.

[28] M. Moriconi, X. Qian, and R. Riemenschneider. "Correct architecture refinement". pp 356–372, 1995.

[29] Eric Newcomer. *Understanding Web Services*. Addision-Wesleyu, September 2002.

[30] M. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. Kramer. "Service Oriented Computing Research Roadmap". *In Dagstuhl Seminar Proceedings* 05462 (SOC), 2006.

[31] M. P. Papazoglou and G. Schlageter, editors. *Cooperative Information Systems: Trends and Directions*. Academic Press, Boston, 1998.

[32] J. Robbins, D. Hilbert, and D. Redmiles. "Extending Design Environments to Software Architecture Design". *In Proc. of the 1996 Knowledge-Based Software Engineering Conference*, pp 63–72, 1996.

[33] J. Robbins and D. Redmiles. "Software architecture design from the perspective of human cognitive needs". *In Proc. of the California Software Symposium*, pp 63–72, 1996.

[34] M.V.S S.Andrew. Tanenbaum, editor. "Distributed Systems: Principles and Paradigms". *Prentice Hall PTR*, 2001.

[35] R.Solheim Skogan, D.Groenmo. "Web service composition in UML". *IEEE international*, pp 47–57, Sept 2004.

[36] S. Vestal. "MetaH Programmer's Manual, Version 1.09". Tec*hnical report, Honeywell Technology Center*, 1996.

[37] Roger Wolter. "Xml Web Services Basics". *Technical report, Microsoft Corporation*, June 2001.

Appendix A

Generic Maude Configurations and the Banking Case Study.

A.1 Component Generic Configuration

The specific Maude-based component generic configuration is depicted in FigureA.1.

```
mod CMP_GNR is
   sorts Msg Event CMPid Property Properties .
   sorts StatCMP obs_StatCMP loc_StatCMP ConfCMP obs_ConfCMP loc_ConfCMP
   subsorts obs_ConfCMP loc_ConfCMP < ConfCMP .
   subsort obs_StatCMP loc_StatCMP < StatCMP .
   subsorts Msg Event StatCMP < ConfCMP .
   subsort Property < Properties .
   sorts obs_Msg loc_Msg obs_Prop loc_Prop .
   subsorts obs_Msg loc_Msg < Msg .
   subsorts obs_Prop loc_Prop < Property .
   subsorts obs_Msg Event obs_StatCMP < obs_ConfCMP .
   subsorts loc_Msg loc_StatCMP < loc_ConfCMP .

   op nul : -> ConfCMP .
   op _ , _ : Properties Properties -> Properties [assoc comm] .
   op <_ | _> : CMPid loc_Prop -> loc_StatCMP .
   op <_ | _> : CMPid obs_Prop -> obs_StatCMP .
   op _ _ : ConfCMP ConfCMP -> ConfCMP [ctor config assoc comm] .

   var I : CMPid .
   var prs1 prs2 : Properties .
   var CfCmp : ConfCMP .

   rl [SplitAT] : < I | prs1 , prs2 > => < I | prs1 > < I | prs2 > .
   rl [RecombineAT] : < I | prs1 > < I | prs2 > => < I | prs1 , prs2 > .
   rl [RemoveNul] : nul CfCmp => CfCmp .
endm
```

Figure A.1: The Maude-based generic patterns of components

A.2 Interface Generic Configuration

The specific Maude-based interface generic configuration is depicted in FigureA.2.

INTF_GENERIC.maude

```
mod INTF_GNR is
   inc CMP_GNR .
   protecting BOOL .
   sorts Intf_NM EX_ConfIntf ConfINTF lid lidL .
   subsorts obs_Msg Event obs_StatCMP < ConfINTF < obs_ConfCMP .
   subsort CMPid < lid .
   subsort lid < lidL .

   op nil : -> lidL .
   op _°_ : lidL lidL -> lidL [assoc comm id: nil] .
   op nul : -> ConfINTF .
   op [_ | _] : Intf_NM ConfINTF -> EX_ConfIntf .
   op _ _ : ConfINTF  ConfINTF -> ConfINTF [ctor config assoc comm] .
   op subsume(_, _) : ConfINTF Intf_NM -> EX_ConfIntf .
   op belong(_, _) : lid lidL -> Bool .
   op weaveCfIntf(_, _) : EX_ConfIntf ConfCMP -> ConfCMP .
   op intercept(_, _) : ConfINTF ConfCMP -> ConfCMP .

   var Cfintf : ConfINTF .
   var Cfcmp : ConfCMP .
   var INM : Intf_NM .
   vars l1, l2 : lid .
   var lL : lidL .
   rl [belong] : belong(l1,l2 ° lL) => (if l1 == l2 then true else if lL == nil then false else belong(l1, lL) fi fi) .
   rl [Subsume] : subsume(Cfintf, INM) => [INM | Cfintf] .
   rl [weaveCfIntf] : weaveCfIntf([INM | Cfintf], Cfcmp) => Cfintf Cfcmp .
   rl [intercept] : intercept(Cfintf, Cfintf Cfcmp) => Cfcmp .
endm
```

Figure A.2: The Maude-based generic patterns of interfaces

A.3 Coordination Generic Configuration

The specific Maude-based coordination generic configuration is depicted in FigureA.3.

ASP_COORD_GENERIC.maude

```
mod ASP_COORD is
   inc INTF_GNR .
   sorts Coord_NM PartnerId PartnerIds CoorOperation CoorOperations .
   subsort PartnerIds < Iid .
   subsort CoorOperation < CoorOperations .
   sorts ObjCoord  ConfCoord Attribute Attributes PartnerAttrs ObjsCoord .
   subsort obs_Msg Event ObjCoord < ConfCoord .
   subsorts Attribute < Attributes .
   subsorts PartnerId < PartnerIds .
   subsorts PartnerIds < PartnerAttrs .
   subsort ObjCoord < ObjsCoord .
   subsort ObjsCoord EX_ConfIntf < ConfCoord .

   op nilAttr : -> Attributes .
   op noneCfCoord : -> ConfCoord .
   op _ $ _ : PartnerIds PartnerIds -> PartnerIds [assoc comm] .
   op _ ; _ : Attributes Attributes -> Attributes [assoc comm] .
   op _ . _ : CoorOperations CoorOperations -> CoorOperations [assoc comm] .
   op _ @ _ : PartnerIds Attributes -> PartnerAttrs .
   op [_ || _] : Coord_NM PartnerAttrs -> ObjCoord .
   op extractCfIntf(_, _) :  ConfCoord Intf_NM -> EX_ConfIntf .
   op _ & _ : ConfCoord ConfCoord -> ConfCoord [assoc comm] .

   vars Cfl1, Cfl2 : ConfINTF .
   var Inm : Intf_NM .
   var CfCt : ConfCoord .

   rl[Split_CfIntf] : [Inm | Cfl1 Cfl2] => [Inm | Cfl1] & [Inm | Cfl2] .
   rl[Recombin_CfIntf] : [Inm | Cfl1] & [Inm | Cfl2] => [Inm | Cfl1 Cfl2] .
   rl[extractCfIntf] : extractCfIntf([Inm | Cfl1] & CfCt, Inm) => [Inm | Cfl1] .
endm
```

Figure A.3: The Maude-based generic pattern for ECA-driven coordinations

A.4 The Aspectual Maude-based Implementation for Banking Case Study

A.4.1 The Customer And Account Component in the Banking Example

In banking case study, the customer and account are required as partners. At the component level, the specific customer component is depicted in Figure B.1, and the specific account component is depicted in Figure A.5.

CUS_CMP_GNR.maude

```
mod CUS_CMP is
   protecting STRING .
   protecting INT .
   inc CMP_GNR .
   sorts WDR CHGADR CustCf CustId .
   subsort WDR < Event .
   subsorts CHGADR < loc_Msg .
   subsort CustCf < ConfCMP .
   subsort CustId < CMPid .

   op Wdr(_,_) : CustId Int -> WDR [ctor] .
   op ChgAdr(_,_) : CustId String -> CHGADR [ctor] .
   op Name: _ : String -> loc_Prop  .
   op Adr: _ : String -> loc_Prop  .

   var CS : CustId .
   var Anew Aold : String .
    rl [chgAdr] : ChgAdr( CS, Anew ) < CS | Adr: Aold >  => < CS | Adr: Anew >
endm
```

Figure A.4: The Maude-based component for customer

The required customer and account interfaces for withdraw

Their corresponding interfaces for withdrawals (both standard withdrawal and Vip withdrawal) are depicted in Figure A.6 and A.7.

A.4.2 Maude-based ECA-driven Interaction for Withdrawal

The Maude-based ECA-driven interaction for withdrawal (both Std- and Vip-withdrawal) is depicted in Figure A.8.

ACNT_CMP_GNR.maude

```
mod ACNT_CMP is
   protecting INT .
   inc CMP_GNR .
   sorts CRDT DBT CHGL TRS His HisL AcntCf AcntId .
   subsorts CRDT DBT TRS < obs_Msg .
   subsort  CHGL < loc_Msg .
   subsorts His < HisL < loc_Msg .
   subsort AcntCf < ConfCMP .
   subsort AcntId < CMPid .
   op Crd(_, _) : AcntId Int -> CRDT [ctor] .
   op Db(_, _) : AcntId Int -> DBT [ctor] .
   op ChgL(_, _) : AcntId Int -> CHGL [ctor] .
   op Trs(_, _, _) : AcntId AcntId Int -> TRS .
   op bal: _: Int -> obs_Prop [ctor gather (&)] .
   op limt: _ : Int -> loc_Prop [ctor gather (&)] .
   op [] : -> His .
   op [_, _] : Int Nat -> His .
   op _, _ : His HisL -> HisL .

   vars A A1 : AcntId .
   vars B B1 L L1 : Int .
   var M : Nat .
   rl [credit] : Crd ( A, M ) < A | bal: B > => < A | bal: B + M > .
   rl [debit] : Db (A, M) < A | bal: B > => < A | bal: B - M > .
   rl [chgl] : ChgL ( A, L1 ) < A | limt: L > => < A | limt: L1 > .
   rl [transfer] : Trs(A, A1, M) < A | bal: B > < A1 | bal: B1 >
                => < A | bal: B - M > < A1 | bal: B1 + M > .
endm
```

Figure A.5: The Maude-based component for account

CUS_INTF4WDR_GNR.maude

```
mod CUS_INTF_GNR is
   inc INTF_GNR .
   protecting INT .
   sorts WDR CSIntf CustId .
   subsort WDR < Event .
   subsort CSIntf < Intf_NM .
   subsort CustId < Iid .

   op CUST : -> Intf_NM .
   op Wdr(_,_) : CustId Int -> WDR [ctor] .
   op getCfIntfwdr(_,_) : ConfCMP IidL -> ConfCMP .
   op getCfIntfwdrf(_) : ConfCMP -> ConfINTF .

   var CS : CustId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var M : Int .
   var CusL : IidL .

   rl [getCfIntfwdr] : getCfIntfwdr(Wdr(CS, M) Cfcp, CusL)
                => (if belong(CS, CusL)
                    then Wdr(CS, M) getCfIntfwdr(Cfcp, CusL)
                    else getCfIntfwdr(Cfcp, CusL)
                    fi) .
   rl [getCfIntfwdrf] : getCfIntfwdrf(Cfcpf getCfIntfwdr(Cfcp, CusL)) => Cfcpf .
endm
```

Figure A.6: The Maude-based interface for customer

ACNT_INTF4WDR_GNR.maude

```
mod ACNT_INTF_GNR is
   inc INTF_GNR .
   protecting INT .
   sorts DBT ACIntf AcntId .
   subsorts DBT < obs_Msg .
   subsort ACIntf < Intf_NM .
   subsort AcntId < Id .

   op ACNT : -> Intf_NM .
   op Db(_, _) : AcntId Int -> DBT [ctor] .
   op bal: _ : Int -> obs_Prop [ctor gather (&)] .
   op getCfIntfbal(_, _) : ConfCMP IdL -> ConfCMP .
   op getCfIntfbalf(_) : ConfCMP -> ConfINTF .
   op pendDB(_) : ConfCMP -> ConfCMP .
   op pendDBf(_) : ConfCMP -> ConfINTF .
   op backDB(_, _) : ConfINTF ConfCMP -> ConfCMP .

   var AC : AcntId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var B : Int .
   var AcntsL : IdL .
   rl [getCfIntfbal] : getCfIntfbal(< AC | bal: B > Cfcp, AcntsL)
                    => (if belong(AC, AcntsL)
                        then < AC | bal: B > getCfIntfbal(Cfcp, AcntsL)
                        else getCfIntfbal(Cfcp, AcntsL) fi) .
   rl [getCfIntfbalf] : getCfIntfbalf(Cfcpf getCfIntfbal(Cfcp, AcntsL)) => Cfcpf .

   rl [pendDB] : pendDB(Db(AC, B) Cfcp) =>  Db(AC, B) pendDB(Cfcp) .
   rl [pendDBf] : pendDBf(Cfcpf pendDB(Cfcp)) => Cfcpf .
   rl [backDB] : backDB(Cfcpf, Cfcp) => Cfcpf Cfcp .
endm
```

Figure A.7: The Maude-based account interface for withdrawal

ASP_COORD_WdrSdtVip_GNR.maude

```
mod COORD_WdrStdVip is
   inc ASP_COORD .
   inc ACNT_INTF_GNR .
   inc CUS_INTF_GNR .
   subsorts AcntId CustId < PartnerIds .

   op WdrSdt : -> Coord_NM .
   op WdrVip : -> Coord_NM .
   op crd: _ : Int -> Attribute [ctor gather (&)] .
   vars CS AC : PartnerIds .
   var M : Nat .
   vars B C : Int .

   crl [WdrSdt] : [WdrSdt || CS $ AC] & [CUST | Wdr(CS, M)] & [ACNT | < AC | bal: B >]
               => [WdrSdt || CS $ AC] & [ACNT | Db(AC, M) < AC | bal: B >]
                  & [CUST | nul]   if B >= M .

   crl [WdrVip] : [WdrVip || (CS $ AC) @ crd: C] & [ CUST | Wdr(CS, M) ] & [ ACNT | < AC | bal: B > ]
               => [WdrVip || (CS $ AC) @ crd: C] & [ ACNT | < AC | bal: B > Db(AC, M)]
                  & [CUST | nul]   if (B + C) >= M .
endm
```

Figure A.8: The Maude-based ECA-driven coordination for withdrawal

A.4.3 Dynamic Weaving Using Maude Reflective Strategy

In Figure A.9 we depict a component specific strategy to control the performance of withdrawal process.

```
ASP_WDR_STR.maude

mod ASP_WDR_Str is
  inc ACNT_INTF_CONF_DW .
  inc CUS_INTF_CONF_DW .
  protecting META-LEVEL .
  vars SplitAT? RecombineAT? RemoveNul? debit? belong? pendDB?
       pendDBf? backDB? getCfIntfwdr? getCfIntfwdrf? getCfIntfbal?
       getCfIntfbalf? intercept? weaveCfIntf? Subsume? Split_CfIntf?
       Recombin_CfIntf? extractCfIntf? WdrSdt? WdrVip? : [Result4Tuple] .
  var T : Term .
  var N : Nat .
  op Compute : Term  Nat -> Term .

  ceq Compute(T, N)
    = if(N == 1) then
        (if(SplitAT? :: Result4Tuple)
         then Compute(getTerm(SplitAT?), N)
         else if(pendDB? :: Result4Tuple)
              then Compute(getTerm(pendDB?),N)
         else if(pendDBf? :: Result4Tuple)
              then Compute(getTerm(pendDBf?),N)
         else if(belong? :: Result4Tuple)
              then Compute(getTerm(belong?),N)
         else if(getCfIntfwdr? :: Result4Tuple)
              then Compute(getTerm(getCfIntfwdr?),N)
         else if(getCfIntfwdrf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfwdrf?),N)
         else if(getCfIntfbal? :: Result4Tuple)
              then Compute(getTerm(getCfIntfbal?),N)
         else if(getCfIntfbalf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfbalf?),N)
         else if(intercept? :: Result4Tuple)
              then Compute(getTerm(intercept?),N)
         else if(Subsume? :: Result4Tuple)
              then Compute(getTerm(Subsume?), N)
         else if(Split_CfIntf? :: Result4Tuple)
              then Compute(getTerm(Split_CfIntf?),N)
         else if(WdrSdt? :: Result4Tuple)
              then Compute(getTerm(WdrSdt?), N)
         else if(WdrVip? :: Result4Tuple)
              then Compute(getTerm(WdrVip?), N)
         else Compute(T, 0)
        fi  fi  fi  fi  fi  fi  fi  fi  fi  fi  fi  fi  fi)

    else
        (if (Recombin_CfIntf? :: Result4Tuple)
         then Compute(getTerm(Recombin_CfIntf?), 0)
         else if extractCfIntf? :: Result4Tuple
              then Compute(getTerm(extractCfIntf?), 0)
         else if (weaveCfIntf? :: Result4Tuple)
              then (if(debit? :: Result4Tuple)
                    then Compute(getTerm(debit?),0)
                    else Compute(getTerm(weaveCfIntf?),0)
                    fi)
         else if(backDB? :: Result4Tuple)
              then Compute(getTerm(backDB?),0)
         else if(RemoveNul? :: Result4Tuple)
              then Compute(getTerm(RemoveNul?),0)
         else if(RecombineAT? :: Result4Tuple)
              then Compute(getTerm(RecombineAT?),0)
         else T
        fi  fi  fi  fi  fi  fi  fi)
  fi

  if SplitAT? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'SplitAT,none,0,unbounded,0)
  /\ RecombineAT? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'RecombineAT,none,0,unbounded,0)
  /\ RemoveNul? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'RemoveNul,none,0,unbounded,0)
  /\ debit? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'debit,none,0,unbounded,0)
  /\ belong? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'belong,none,0,unbounded,0)
  /\ pendDB? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'pendDB,none,0,unbounded,0)
  /\ pendDBf? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'pendDBf,none,0,unbounded,0)
  /\ backDB? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'backDB,none,0,unbounded,0)
  /\ getCfIntfwdr? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'getCfIntfwdr,none,0,unbounded,0)
  /\ getCfIntfwdrf? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'getCfIntfwdrf,none,0,unbounded,0)
  /\ getCfIntfbal? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'getCfIntfbal,none,0,unbounded,0)
  /\ getCfIntfbalf? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'getCfIntfbalf,none,0,unbounded,0)
  /\ intercept? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'intercept,none,0,unbounded,0)
  /\ weaveCfIntf? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'weaveCfIntf,none,0,unbounded,0)
  /\ Subsume? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'Subsume,none,0,unbounded,0)
  /\ Split_CfIntf? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'Split_CfIntf,none,0,unbounded,0)
  /\ WdrSdt? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'WdrSdt,none,0,unbounded,0)
  /\ WdrVip? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'WdrVip,none,0,unbounded,0)
  /\ Recombin_CfIntf? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'Recombin_CfIntf,none,0,unbounded,0)
  /\ extractCfIntf? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
              T,'extractCfIntf,none,0,unbounded,0) .
  eq Compute(T, N) = T [owise] .
endm
```

Figure A.9: The reflective strategy implementing the aspectual dynamic adaptive coordination

Appendix B

The Aspectual Maude-based Implementation for E-commerce CaseS

B.1. All the Components Involved In the E-commerce System

Figure B.1 depicts the complete specification of customer component.

CUS_CMP_GNR.maude

```
mod CUS_CMP is
  protecting STRING .
  protecting RAT .
  inc CMP_GNR .
  sort Customer .
  sorts ORDP Ask4B Ask4D Ask4P GB SuccP Cancel CHGADR CustCf CustId PrId Prof Shipment .
  subsorts ORDP Ask4B Ask4D Ask4P GB SuccP < Event .
  subsort CHGADR < loc_Msg .
  subsort CustCf < ConfCMP .
   subsorts CustId PrId < CMPid .

  ops Normal Silver Golden : -> Prof .
  ops TakeYself Normal Express : -> Shipment .
  op OrderPr(_, _ ,_) : CustId PrId Int -> ORDP .
  op ask4Bill(_) : CustId -> Ask4B .
  op ask4Deliver(_, _) : CustId Shipment -> Ask4D .
  op ask4Pay(_) : CustId -> Ask4P .
  op succPay(_) : CustId -> SuccP .
  op getBill(_, _) : CustId Rat -> GB .
  op cancelPr(_, _, _) : CustId PrId Int -> Cancel .
  op ChgAdr(_,_) : CustId String -> CHGADR .
  op Name: _ : String -> loc_Prop .
  op Adr: _ : String -> loc_Prop .
  op Profile: _ : Prof -> obs_Prop .

  var CS : CustId .
  vars Anew Aold : String .
  rl [chgAdr] : ChgAdr( CS, Anew ) < CS | Adr: Aold > => < CS | Adr: Anew > .
endm
```

Figure B.1: The Maude-based customer component

The complete specification of product component is depicted in Figure B.2. In Figure B.3 we depict the complete specification of shopping card component.

PROD_CMP_GNR.maude

```
mod PROD_CMP is
   protecting STRING .
   protecting INT .
   inc CMP_GNR .
   sorts ProdCf PrId UPDPr .
   subsort ProdCf < ConfCMP .
   subsort PrId < CMPid .
   subsort UPDPr < obs_Msg .

  op AvailQt: _ : Nat -> obs_Prop .
  op Price: _ : Int -> loc_Prop .
  op UpdatePr(_, _) : PrId Int -> UPDPr .

  var Pid : PrId .
  vars M N : Int .
  rl[UpdatePr] : UpdatePr(Pid, M) < Pid | AvailQt: N > => < Pid | AvailQt: (N - M) > .
endm
```

Figure B.2: The Maude-based product component

SHC_CMP_GNR.maude

```
mod SHC_CMP is
    protecting INT .
    inc CMP_GNR .
    sorts ShCCf CdId CustId PrId ADP2B List .
    subsort ShCCf < ConfCMP .
    subsorts CdId CustId PrId < CMPid .
    subsort ADP2B < obs_Msg .

    op [] : -> List [ctor] .
    op [_, _] : PrId Int -> List .
    op _ ° _ : List List -> List [assoc comm] .
    op CusId: _ : CustId -> obs_Prop .
    op Basket: _ : List -> obs_Prop .
    op AddPr2Bask(_, _) : CdId List -> ADP2B .

    var Pid : PrId .
    var Cid : CdId .
    vars L  L1 : List .
    vars M N : Int .
    rl[AddPr2Bask] : AddPr2Bask(Cid, L1) < Cid | Basket: L > => < Cid | Basket: (L ° L1) > .
    rl[CombineList] : [Pid, M] ° [Pid, N] => [Pid, M + N] .
    rl[DropNullList] : [] ° L => L .
 endm
```

Figure B.3: The Maude-based shopping card component

In the right-hand side of Figure B.4 the complete specification of history is depicted, and in the left-hand side the Maude module *DATE* is shown.

DATE.maude

```
mod DATE is
  protecting INT .
  protecting BOOL .
  sorts Date Day Month Year .
  subsorts Day Month Year < Int .

  op Today : -> Date .
  op [_, _, _] : Day Month Year -> Date .
  op getDate(_, _, _) : Int Int Int -> Date .
  op GreaterM(_, _, _) : Date Date Int -> Bool .

  vars Dc Mc Yc D M Y Mt : Int .
  eq GreaterM(getDate(Dc, Mc, Yc), getDate(D, M, Y), Mt)
   = if(((Yc == Y) and ((Mc - M) > Mt)) or ((Yc > Y)
        and (((12 + Mc) - M) > Mt)) or ((Yc == Y) and
        ((Mc - M) == Mt) and (Dc > D)))
      then true
      else false
      fi .
  rl [getDate] : getDate(D, M, Y) => [D, M, Y] .
endm
```

HIS_CMP_GNR.maude

```
mod HIS_CMP is
   protecting STRING .
   protecting RAT .
   protecting DATE .
   inc CMP_GNR .
   sorts HisCf CustId HisId UPDSP GCDAY .
   subsort HisCf < ConfCMP .
   subsorts CustId HisId < CMPid .
   subsort UPDSP < obs_Msg .
   subsort GCDAY < Event .

   op CusId: _ : CustId -> obs_Prop .
   op Spending: _ : Rat -> obs_Prop .
   op LastUpdateSP: _ : Date -> obs_Prop .
   op UpdateSp(_, _) : CustId Rat -> UPDSP [ctor] .
   op bill: _ : Rat -> obs_Prop .
   op getCurrentDay(_, _) : HisId Date -> GCDAY [ctor] .

   var CId : CustId .
   var Hid : HisId .
   var M S : Rat .
   rl[UpdateSp] : UpdateSp(CId, M) < Hid | CusId: CId, Spending: S >
                  => < Hid | CusId: CId, Spending: (S + M) > .
endm
```

Figure B.4: The Maude-based history component

The Maude-based bank card component is defined in Figure B.5.

BCARD_CMP_GNR.maude

```
mod BCARD_CMP is
   protecting INT .
   inc CMP_GNR .
   sort BCardCf .
   subsort BCardCf < ConfCMP .
   sorts AcntId BCardId .
   subsort AcntId BCardId < CMPid .

   op AcntNr: _ : AcntId -> obs_Prop [ctor gather (&)] .
endm
```

Figure B.5: The Maude-based bank-card component

B.1.1 The Order Activity

The required interfaces for order activity

In the order activity components—customer, product and shopping card—are involved at the component level, below we depict the specific interfaces of these involved components. In Figure B.6 we depict the complete specification of product interface for order, in Figure B.7 the shopping card interface is depicted, and in Figure B.8 we depict the specific customer interface for order activity.

Prod_INTF4ORD_GNR.maude

```
mod PROD_INTF4ORD_GNR is
   inc INTF_GNR .
   protecting INT .
   sorts ProdIntf PrId .
   subsort ProdIntf < Intf_NM .
   subsort PrId < Id .

   op PROD : -> Intf_NM .
   op AvailQt: _ : Int -> obs_Prop .
   op getCfIntfAQt(_, _) : ConfCMP IdL -> ConfCMP .
   op getCfIntfAQtf(_) : ConfCMP -> ConfINTF .

   var Pr : PrId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var B : Int .
   var PrIdL : IdL .

   rl [getCfIntfAQt] : getCfIntfAQt(< Pr | AvailQt: B > Cfcp, PrIdL)
                    => (if belong(Pr, PrIdL)
                        then < Pr | AvailQt: B > getCfIntfAQt(Cfcp, PrIdL)
                        else getCfIntfAQt(Cfcp, PrIdL) fi) .
   rl [getCfIntfAQtf] : getCfIntfAQtf(Cfcpf getCfIntfAQt(Cfcp, PrIdL)) => Cfcpf .
endm
```

Figure B.6: The Maude-based product for order interface

SHC_INTF4ORD_GNR.maude

```
mod SHC_INTF4ORD_GNR is
  inc INTF_GNR .
  protecting INT .
  sorts AP2B ShCIntf CdId PrId CustId List .
  subsorts AP2B < obs_Msg .
  subsort ShCIntf < Intf_NM .
  subsort CdId PrId < Iid .

   op [] : -> List [ctor] .
   op [_, _] : PrId Int -> List .
   op _ ° _ : List List -> List [assoc comm] .
   op SHC : -> Intf_NM .
   op AddPr2Bask(_, _) : CdId List -> AP2B .
   op CusId: _ : CustId -> obs_Prop .
   op Basket: _ : List -> obs_Prop .
   op getCfIntfBasket(_, _) : ConfCMP IidL -> ConfCMP .
   op getCfIntfBasketf(_) : ConfCMP -> ConfINTF .
   op pendAddPr2Bask(_) : ConfCMP -> ConfCMP .
   op pendAddPr2Baskf(_) : ConfCMP -> ConfINTF .
   op backAddPr2Bask(_, _) : ConfINTF ConfCMP -> ConfCMP .

   var Cid : CdId .
   var Cus : CustId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var L : List .
   var CIdL : IidL .
   rl [getCfIntfBasket] : getCfIntfBasket(< Cid | CusId: Cus > < Cid | Basket: L > Cfcp, CIdL)
                       => (if belong(Cid, CIdL)
                           then < Cid | CusId: Cus > < Cid | Basket: L > getCfIntfBasket(Cfcp, CIdL)
                           else getCfIntfBasket(Cfcp, CIdL) fi) .
  rl [getCfIntfBasketf] : getCfIntfBasketf(Cfcpf getCfIntfBasket(Cfcp, CIdL)) => Cfcpf .

  rl [pendAddPr2Bask] : pendAddPr2Bask(AddPr2Bask(Cid, L) Cfcp) => AddPr2Bask(Cid, L) pendAddPr2Bask(Cfcp) .
  rl [pendAddPr2Baskf] : pendAddPr2Baskf(Cfcpf pendAddPr2Bask(Cfcp)) => Cfcpf .
  rl [backAddPr2Bask] : backAddPr2Bask(Cfcpf, Cfcp) => Cfcpf Cfcp .
endm
```

Figure B.7: The Maude-based shopping card for order interface

CUS_INTF4ORD_GNR.maude

```
mod CUS_INTF4ORD_GNR is
  inc INTF_GNR .
  protecting INT .
  sorts ORDP CSIntf CustId PrId .
  subsort ORDP < Event .
  subsort CSIntf < Intf_NM .
  subsort CustId PrId < Id .

  op CUST : -> Intf_NM .
 op OrderPr(_, _, _) : CustId PrId Int -> ORDP [ctor] .
  op getCfIntfOrdPr(_, _, _) : ConfCMP IdL IdL -> ConfCMP .
  op getCfIntfOrdPrf(_) : ConfCMP -> ConfINTF .

  var CS : CustId .
  var Pr : PrId .
  var Cfcp : ConfCMP .
  var Cfcpf : ConfINTF .
  var M : Int .
  vars CusL PrL : IdL .

  rl [getCfIntfOrdPr] : getCfIntfOrdPr(OrderPr(CS, Pr, M) Cfcp, CusL, PrL)
                     => (if (belong(CS, CusL) and belong(Pr, PrL))
                         then OrderPr(CS, Pr, M) getCfIntfOrdPr(Cfcp, CusL, PrL)
                         else getCfIntfOrdPr(Cfcp, CusL, PrL) fi) .
  rl [getCfIntfOrdPrf] : getCfIntfOrdPrf(Cfcpf getCfIntfOrdPr(Cfcp, CusL, PrL)) => Cfcpf .
endm
```

Figure B.8: The Maude-based customer for order interface

The Maude-implementation of the order ECA-rule

Based on the above interfaces, the Maude-based implementation of ECA-rule order is depicted in Figure B.9.

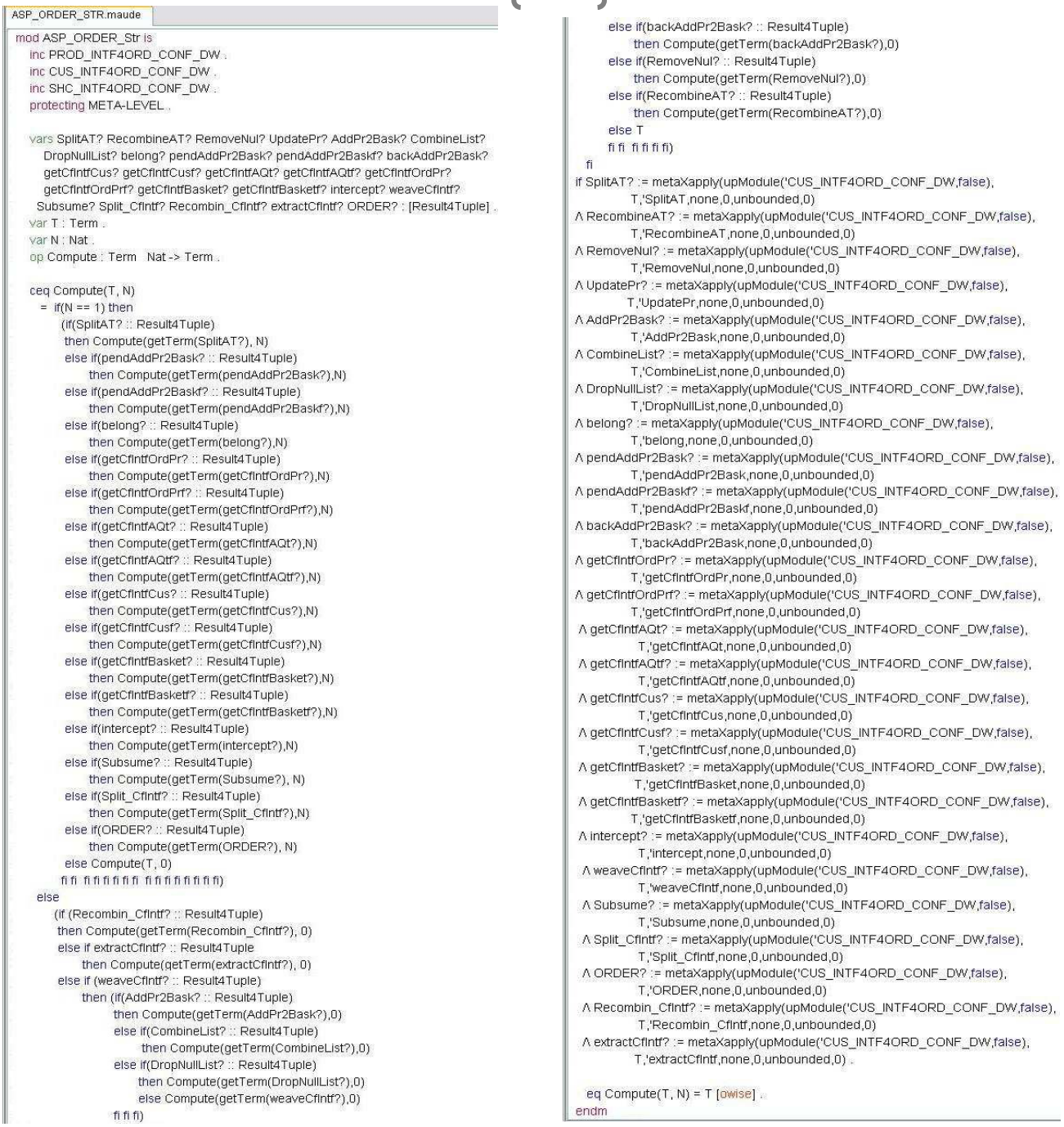
ASP_COORD_ORDER_GNR.maude

```
mod COORD_ORDER is
  inc ASP_COORD .
  inc PROD_INTF4ORD_GNR .
  inc CUS_INTF4ORD_GNR .
  inc SHC_INTF4ORD_GNR .
  subsorts Prld Custld Cdld < Partnerlds .

  op ORDER : -> Coord_NM .

  vars CS Pr Cd  : Partnerlds .
  var M B : Int .
  var L : List .

  crl [ORDER] : [ORDER || CS $ Pr $ Cd] & [CUST | OrderPr(CS, Pr, M)] & [PROD | < Pr | AvailQt: B >]
                & [SHC | < Cd | Basket: L >] & [SHC | < Cd | Cusld: CS >]
             => [ORDER || CS $ Pr $ Cd] & [SHC | AddPr2Bask(Cd, [Pr, M]) < Cd | Basket: L >]
                & [SHC | < Cd | Cusld: CS >] & [PROD | < Pr | AvailQt: B >] & [CUST | nul] if B >= M .
endm
```

Figure B.9: The Maude-based order coordination

Dynamic weaving of the Order process as Maude strategy

The complete specific strategy depicted in Figure B.10 is to control the process of order activity.



```
ASP_ORDER_STR.maude
mod ASP_ORDER_Str is
  inc PROD_INTF4ORD_CONF_DW .
  inc CUS_INTF4ORD_CONF_DW .
  inc SHC_INTF4ORD_CONF_DW .
  protecting META-LEVEL .

  vars SplitAT? RecombineAT? RemoveNul? UpdatePr? AddPr2Bask? CombineList?
    DropNullList? belong? pendAddPr2Bask? pendAddPr2Baskf? backAddPr2Bask?
    getCfIntfCus? getCfIntfCusf? getCfIntfAQt? getCfIntfAQtf? getCfIntfOrdPr?
    getCfIntfOrdPrf? getCfIntfBasket? getCfIntfBasketf? intercept? weaveCfIntf?
   Subsume? Split_CfIntf? Recombin_CfIntf? extractCfIntf? ORDER? : [Result4Tuple] .
  var T : Term .
  var N : Nat .
  op Compute : Term   Nat -> Term .

  ceq Compute(T, N)
    =  if(N == 1) then
        (if(SplitAT? :: Result4Tuple)
         then Compute(getTerm(SplitAT?), N)
         else if(pendAddPr2Bask? :: Result4Tuple)
              then Compute(getTerm(pendAddPr2Bask?),N)
         else if(pendAddPr2Baskf? :: Result4Tuple)
              then Compute(getTerm(pendAddPr2Baskf?),N)
         else if(belong? :: Result4Tuple)
              then Compute(getTerm(belong?),N)
         else if(getCfIntfOrdPr? :: Result4Tuple)
              then Compute(getTerm(getCfIntfOrdPr?),N)
         else if(getCfIntfOrdPrf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfOrdPrf?),N)
         else if(getCfIntfAQt? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAQt?),N)
         else if(getCfIntfAQtf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAQtf?),N)
         else if(getCfIntfCus? :: Result4Tuple)
              then Compute(getTerm(getCfIntfCus?),N)
         else if(getCfIntfCusf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfCusf?),N)
         else if(getCfIntfBasket? :: Result4Tuple)
              then Compute(getTerm(getCfIntfBasket?),N)
         else if(getCfIntfBasketf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfBasketf?),N)
         else if(intercept? :: Result4Tuple)
              then Compute(getTerm(intercept?),N)
         else if(Subsume? :: Result4Tuple)
              then Compute(getTerm(Subsume?), N)
         else if(Split_CfIntf? :: Result4Tuple)
              then Compute(getTerm(Split_CfIntf?),N)
         else if(ORDER? :: Result4Tuple)
              then Compute(getTerm(ORDER?), N)
         else Compute(T, 0)
        fi fi  fi fi fi fi fi  fi fi fi fi fi fi fi fi)
    else
       (if (Recombin_CfIntf? :: Result4Tuple)
        then Compute(getTerm(Recombin_CfIntf?), 0)
        else if extractCfIntf? :: Result4Tuple
             then Compute(getTerm(extractCfIntf?), 0)
        else if (weaveCfIntf? :: Result4Tuple)
             then (if(AddPr2Bask? :: Result4Tuple)
                   then Compute(getTerm(AddPr2Bask?),0)
                   else if(CombineList? :: Result4Tuple)
                        then Compute(getTerm(CombineList?),0)
                   else if(DropNullList? :: Result4Tuple)
                        then Compute(getTerm(DropNullList?),0)
                        else Compute(getTerm(weaveCfIntf?),0)
                   fi fi fi)
        else if(backAddPr2Bask? :: Result4Tuple)
             then Compute(getTerm(backAddPr2Bask?),0)
        else if(RemoveNul? :: Result4Tuple)
             then Compute(getTerm(RemoveNul?),0)
        else if(RecombineAT? :: Result4Tuple)
             then Compute(getTerm(RecombineAT?),0)
        else T
        fi fi  fi fi fi fi)
    fi
  if SplitAT? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'SplitAT,none,0,unbounded,0)
  /\ RecombineAT? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'RecombineAT,none,0,unbounded,0)
  /\ RemoveNul? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'RemoveNul,none,0,unbounded,0)
  /\ UpdatePr? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
             T,'UpdatePr,none,0,unbounded,0)
  /\ AddPr2Bask? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'AddPr2Bask,none,0,unbounded,0)
  /\ CombineList? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'CombineList,none,0,unbounded,0)
  /\ DropNullList? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'DropNullList,none,0,unbounded,0)
  /\ belong? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'belong,none,0,unbounded,0)
  /\ pendAddPr2Bask? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'pendAddPr2Bask,none,0,unbounded,0)
  /\ pendAddPr2Baskf? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'pendAddPr2Baskf,none,0,unbounded,0)
  /\ backAddPr2Bask? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'backAddPr2Bask,none,0,unbounded,0)
  /\ getCfIntfOrdPr? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'getCfIntfOrdPr,none,0,unbounded,0)
  /\ getCfIntfOrdPrf? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'getCfIntfOrdPrf,none,0,unbounded,0)
  /\ getCfIntfAQt? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'getCfIntfAQt,none,0,unbounded,0)
  /\ getCfIntfAQtf? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'getCfIntfAQtf,none,0,unbounded,0)
  /\ getCfIntfCus? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'getCfIntfCus,none,0,unbounded,0)
  /\ getCfIntfCusf? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'getCfIntfCusf,none,0,unbounded,0)
  /\ getCfIntfBasket? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'getCfIntfBasket,none,0,unbounded,0)
  /\ getCfIntfBasketf? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'getCfIntfBasketf,none,0,unbounded,0)
  /\ intercept? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'intercept,none,0,unbounded,0)
  /\ weaveCfIntf? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'weaveCfIntf,none,0,unbounded,0)
  /\ Subsume? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'Subsume,none,0,unbounded,0)
  /\ Split_CfIntf? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'Split_CfIntf,none,0,unbounded,0)
  /\ ORDER? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'ORDER,none,0,unbounded,0)
  /\ Recombin_CfIntf? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'Recombin_CfIntf,none,0,unbounded,0)
  /\ extractCfIntf? := metaXapply(upModule('CUS_INTF4ORD_CONF_DW,false),
              T,'extractCfIntf,none,0,unbounded,0) .

   eq Compute(T, N) = T [owise] .
endm
```

Figure B.10: The reflective strategy for order activity

B.1.2 The Confirmation Activity

The required interfaces for confirmation activity

The confirmation activity requires customer, product, shopping card and history as partners. Below we depict their corresponding interfaces. The specific customer interface for confirmation is depicted in Figure B.11, the product interface for confirmation is shown in Figure B.12, the precise shopping card interface is shown in Figure B.13, and the history interface is in Figure B.14.

CUS_INTF4CFM_GNR.maude

```
mod CUS_INTF4CFM_GNR is
  inc INTF_GNR .
  protecting RAT .
  sorts Ask4B GB CSIntf CustId Prof .
  subsort Ask4B GB < Event .
  subsort CSIntf < Intf_NM .
  subsort CustId < Iid .

  op CUST : -> Intf_NM .
 op ask4Bill(_) : CustId -> Ask4B .
  op getBill(_, _) : CustId Rat -> GB .
  ops Normal Silver Golden : -> Prof .
  op Profile: _ : Prof -> obs_Prop .
  op getCfIntfAsk4B(_, _) : ConfCMP IidL -> ConfCMP .
  op getCfIntfAsk4Bf(_) : ConfCMP -> ConfINTF .

  var CS : CustId .
  var Cfcp : ConfCMP .
  var Cfcpf : ConfINTF .
  var CusL : IidL .
  var prf : Prof .
  rl [getCfIntfAsk4B] : getCfIntfAsk4B(ask4Bill(CS) < CS | Profile: prf > Cfcp, CusL)
                    => (if belong(CS, CusL)
                        then ask4Bill(CS) < CS | Profile: prf > getCfIntfAsk4B(Cfcp, CusL)
                        else getCfIntfAsk4B(Cfcp, CusL) fi) .
  rl [getCfIntfAsk4Bf] : getCfIntfAsk4Bf(Cfcpf getCfIntfAsk4B(Cfcp, CusL)) => Cfcpf .
endm
```

Figure B.11: The Maude-based customer for confirmation interface

PROD_INTF4CFM_GNR.maude

```
mod PROD_INTF4CFM_GNR is
   inc INTF_GNR .
   protecting RAT .
   sorts ProdIntf PrId UPDPr .
   subsort ProdIntf < Intf_NM .
   subsort PrId < Id .
   subsorts UPDPr < obs_Msg .

   op PROD : -> Intf_NM .
   op AvailQt: _ : Int -> obs_Prop .
   op Price: _ : Rat -> obs_Prop .
   op UpdatePr(_, _) : PrId Int -> UPDPr [ctor] .
   op getCfIntfAQt(_) : ConfCMP -> ConfCMP .
   op getCfIntfAQtf(_) : ConfCMP -> ConfINTF .

   var Pr : PrId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var B : Int .
   var M : Rat .

   rl [getCfIntfAQt] : getCfIntfAQt(< Pr | AvailQt: B > < Pr | Price: M > Cfcp)
                    => < Pr | Price: M > < Pr | AvailQt: B > getCfIntfAQt(Cfcp) .
   rl [getCfIntfAQtf] : getCfIntfAQtf(Cfcpf getCfIntfAQt(Cfcp)) => Cfcpf .
endm
```

Figure B.12: The Maude-based product for confirmation interface

SHC_INTF4CFM_GNR.maude

```
mod SHC_INTF4CFM_GNR is
   inc INTF_GNR .
   protecting INT .
   sorts ShCIntf CdId PrId CustId List .
   subsort ShCIntf < Intf_NM .
   subsort CdId PrId < Id .

   op [] : -> List [ctor] .
   op [_, _] : PrId Int -> List .
   op _ ° _ : List List -> List [assoc comm] .
   op SHC : -> Intf_NM .
   op CusId: _ : CustId -> obs_Prop .
   op Basket: _ : List -> obs_Prop .
   op getCfIntfBasket(_, _) : ConfCMP IdL -> ConfCMP .
   op getCfIntfBasketf(_) : ConfCMP -> ConfINTF .

   var Cid : CdId .
   var Cus : CustId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var L : List .
   var CIdL : IdL .
   rl [getCfIntfBasket] : getCfIntfBasket(< Cid | CusId: Cus > < Cid | Basket: L > Cfcp, CIdL)
                       => (if belong(Cid, CIdL)
                           then < Cid | CusId: Cus > < Cid | Basket: L > getCfIntfBasket(Cfcp, CIdL)
                           else getCfIntfBasket(Cfcp, CIdL) fi) .
   rl [getCfIntfBasketf] : getCfIntfBasketf(Cfcpf getCfIntfBasket(Cfcp, CIdL)) => Cfcpf .
endm
```

Figure B.13: The Maude-based shopping card for confirmation interface

HIS_INTF4CFM_GNR.maude

```
mod HIS_INTF4CFM_GNR is
   inc INTF_GNR .
   protecting RAT .
   sorts HisIntf HisId CustId .
   subsort HisIntf < Intf_NM .
   subsort HisId CustId < Iid .

   op HIS : -> Intf_NM .
   op CusId: _ : CustId -> obs_Prop  .
   op bill: _ : Rat -> obs_Prop  .
   op getCfIntfSpd(_, _) : ConfCMP IidL -> ConfCMP .
   op getCfIntfSpdf(_) : ConfCMP -> ConfINTF .

   var His : HisId .
   var Cus : CustId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var HisL : IidL .
   var M B : Rat .
   rl [getCfIntfSpd] : getCfIntfSpd(< His | CusId: Cus > < His | bill: B > Cfcp, HisL)
                    => (if belong(His, HisL)
                        then < His | CusId: Cus > < His | bill: B > getCfIntfSpd(Cfcp, HisL)
                        else getCfIntfSpd(Cfcp, HisL) fi) .
  rl [getCfIntfSpdf] : getCfIntfSpdf(Cfcpf getCfIntfSpd(Cfcp, HisL)) => Cfcpf .
endm
```

Figure B.14: The Maude-based history for confirmation interface

The Maude-implementation of the confirmation ECA-rule

The Maude implementation of the ECA-driven interaction rule confirmation is depicted in Figure B.15.

ASP_COORD_CFM_GNR.maude

```
mod COORD_CFM is
   inc ASP_COORD .
   inc PROD_INTF4CFM_GNR .
   inc CUS_INTF4CFM_GNR .
   inc SHC_INTF4CFM_GNR .
   inc HIS_INTF4CFM_GNR .
   subsorts PrId CustId CdId HisId < PartnerIds .
   sort ProfRd .
   subsort ProfRd < Int .

   ops NormalRd SilverRd GoldenRd : -> ProfRd .
   op CFM : -> Coord_NM .
   op discrate: _ : Int -> Attribute [ctor gather (&)] .
   eq NormalRd = 0 .
   eq SilverRd = 5 .
   eq GoldenRd = 10 .

   vars CS Pr Cd His : PartnerIds .
   var M Q R : Int .
   var B P : Rat .
   var L : List .
   var prf : Prof .

   crl [Calculate] : [PROD | < Pr | Price: P >] & [PROD | < Pr | AvailQt: Q >] & [SHC | < Cd | Basket: ([Pr, M] ° L) >]
                     & [SHC | < Cd | CusId: CS >] & [HIS | < His | CusId: CS >] & [HIS | < His | bill: B >]
                  => [PROD | < Pr | Price: P >] & [PROD | UpdatePr(Pr, M) < Pr | AvailQt: Q >] & [SHC | < Cd | Basket: L >]
                     & [SHC | < Cd | CusId: CS >] & [HIS | < His | CusId: CS >] & [HIS | < His | bill: ((M * P) + B) >] if Q >= M .

   rl [CFM] : [CFM || (CS $ Pr $ Cd $ His) @ discrate: R] & [CUST | ask4Bill(CS)] & [HIS | < His | CusId: CS >] & [HIS | < His | bill: B >]
           => [CFM || (CS $ Pr $ Cd $ His) @ discrate: R] & [CUST | getBill(CS, (B * ((100 - R) / 100)))]
              & [HIS | < His | CusId: CS >] & [HIS | < His | bill: (B * ((100 - R) / 100)) >] .
endm
```

Figure B.15: The Maude-based confirmation coordination

Dynamic weaving of the confirmation process as Maude strategy

In Figure B.16 we depict the complete specific strategy to control the performance of confirmation process.

ASP_CFM_STR.maude

```
mod ASP_CFM_Str is
   inc PROD_INTF4CFM_CONF_DW .
   inc CUS_INTF4CFM_CONF_DW .
   inc SHC_INTF4CFM_CONF_DW .
   inc HIS_INTF4CFM_CONF_DW .
   protecting META-LEVEL .

   vars SplitAT? RecombineAT? RemoveNul? UpdatePr? belong? getCfIntfAsk4B?
        getCfIntfAsk4Bf? getCfIntfAQt? getCfIntfAQtf?  getCfIntfBasket? getCfIntfBasketf?
        getCfIntfSpd? getCfIntfSpdf? intercept? weaveCfIntf? Subsume? Split_CfIntf?
        Recombin_CfIntf? extractCfIntf? Calculate? CFM? : [Result4Tuple] .

   var T : Term .
   var N : Nat .
   op Compute : Term  Nat -> Term .

   ceq Compute(T, N)
    = if(N == 1) then
        (if(SplitAT? :: Result4Tuple)
         then Compute(getTerm(SplitAT?), N)
         else if(belong? :: Result4Tuple)
              then Compute(getTerm(belong?),N)
         else if(getCfIntfAsk4B? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAsk4B?),N)
         else if(getCfIntfAsk4Bf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAsk4Bf?),N)
         else if(getCfIntfAQt? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAQt?),N)
         else if(getCfIntfAQtf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAQtf?),N)
         else if(getCfIntfBasket? :: Result4Tuple)
              then Compute(getTerm(getCfIntfBasket?),N)
         else if(getCfIntfBasketf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfBasketf?),N)
         else if(getCfIntfSpd? :: Result4Tuple)
              then Compute(getTerm(getCfIntfSpd?),N)
         else if(getCfIntfSpdf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfSpdf?),N)
         else if(Subsume? :: Result4Tuple)
              then Compute(getTerm(Subsume?), N)
         else if(Split_CfIntf? :: Result4Tuple)
              then Compute(getTerm(Split_CfIntf?),N)
         else if(CFM? :: Result4Tuple)
              then (if (Calculate? :: Result4Tuple)
                    then Compute(getTerm(Calculate?),N)
                    else Compute(getTerm(CFM?), N)
                    fi)
         else Compute(T, 0)
        fi fi fi fi fi fi  fi  fi fi fi fi fi fi)

    else
        (if (Recombin_CfIntf? :: Result4Tuple)
         then Compute(getTerm(Recombin_CfIntf?), 0)
         else if extractCfIntf? :: Result4Tuple
              then Compute(getTerm(extractCfIntf?), 0)
         else if(intercept? :: Result4Tuple)
              then Compute(getTerm(intercept?),0)
         else if (weaveCfIntf? :: Result4Tuple)
              then (if(UpdatePr? :: Result4Tuple)
                    then Compute(getTerm(UpdatePr?),0)
                    else Compute(getTerm(weaveCfIntf?),0)
                    fi)
         else if(RemoveNul? :: Result4Tuple)
              then Compute(getTerm(RemoveNul?),0)
         else if(RecombineAT? :: Result4Tuple)
              then Compute(getTerm(RecombineAT?),0)
         else T
        fi fi  fi fi fi fi fi)
    fi
if SplitAT? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'SplitAT,none,0,unbounded,0)
/\ RecombineAT? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'RecombineAT,none,0,unbounded,0)
/\ RemoveNul? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'RemoveNul,none,0,unbounded,0)
/\ UpdatePr? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'UpdatePr,none,0,unbounded,0)
/\ belong? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'belong,none,0,unbounded,0)
/\ getCfIntfAsk4B? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'getCfIntfAsk4B,none,0,unbounded,0)
/\ getCfIntfAsk4Bf? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'getCfIntfAsk4Bf,none,0,unbounded,0)
/\ getCfIntfAQt? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'getCfIntfAQt,none,0,unbounded,0)
/\ getCfIntfAQtf? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'getCfIntfAQtf,none,0,unbounded,0)
/\ getCfIntfSpd? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'getCfIntfSpd,none,0,unbounded,0)
/\ getCfIntfSpdf? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'getCfIntfSpdf,none,0,unbounded,0)
/\ getCfIntfBasket? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'getCfIntfBasket,none,0,unbounded,0)
/\ getCfIntfBasketf? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'getCfIntfBasketf,none,0,unbounded,0)
/\ intercept? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'intercept,none,0,unbounded,0)
/\ weaveCfIntf? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'weaveCfIntf,none,0,unbounded,0)
/\ Subsume? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'Subsume,none,0,unbounded,0)
/\ Split_CfIntf? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'Split_CfIntf,none,0,unbounded,0)
/\ Calculate? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'Calculate,none,0,unbounded,0)
/\ CFM? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'CFM,none,0,unbounded,0)
/\ Recombin_CfIntf? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'Recombin_CfIntf,none,0,unbounded,0)
/\ extractCfIntf? := metaXapply(upModule('CUS_INTF4CFM_CONF_DW,false),
              T,'extractCfIntf,none,0,unbounded,0) .
 eq Compute(T, N) = T [owise] .
endm
```

Figure B.16: The reflective strategy for confirmation activity

B.1.3 The Cancel Activity

The required interfaces for cancel activity

In cancel activity the customer, product and history components are required at the component level. Their respective interfaces for the cancel activity are depicted in the following figures, e.g. the customer interface is depicted in Figure B.17, the product interface is shown in Figure B.18, and the history interface is shown in Figure B.19.

CUS_INTF4CANCEL_GNR.maude

```
mod CUS_INTF4CANCEL_GNR is
   inc INTF_GNR .
   protecting RAT .
   sorts GB Cancel CSIntf CustId PrId .
   subsort GB Cancel < Event .
   subsort CSIntf < Intf_NM .
   subsort CustId PrId < Iid .

   op CUST : -> Intf_NM .
   op getBill(_, _) : CustId Rat -> GB .
   op cancelPr(_, _, _) : CustId PrId Int -> Cancel .
   op getCfIntfCancel(_, _) : ConfCMP IidL -> ConfCMP .
   op getCfIntfCancelf(_) : ConfCMP -> ConfINTF .

   var CS : CustId .
   var Pr : PrId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var CusL : IidL .
   var B : Rat .
   var M : Int .

   rl [getCfIntfCancel] : getCfIntfCancel(getBill(CS, B) cancelPr(CS, Pr, M) Cfcp, CusL)
                       => (if belong(CS, CusL)
                           then cancelPr(CS, Pr, M) getBill(CS, B) getCfIntfCancel(Cfcp, CusL)
                           else getCfIntfCancel(Cfcp, CusL) fi) .
   rl [getCfIntfCancelf] : getCfIntfCancelf(Cfcpf getCfIntfCancel(Cfcp, CusL)) => Cfcpf .
endm
```

Figure B.17: The Maude-based customer for cancel interface

PROD_INTF4CANCEL_GNR.maude

```
mod PROD_INTF4CANCEL_GNR is
   inc INTF_GNR .
   protecting RAT .
   sorts ProdIntf PrId UPDPr .
   subsort ProdIntf < Intf_NM .
   subsort PrId < Id .
   subsorts UPDPr < obs_Msg .

   op PROD : -> Intf_NM .
   op AvailQt: _ : Int -> obs_Prop .
   op Price: _ : Rat -> obs_Prop .
   op UpdatePr(_, _) : PrId Int -> UPDPr [ctor] .
   op getCfIntfAQt(_) : ConfCMP -> ConfCMP .
   op getCfIntfAQtf(_) : ConfCMP -> ConfINTF .

   var Pr : PrId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var B : Int .
   var M : Rat .

   rl [getCfIntfAQt] : getCfIntfAQt(< Pr | AvailQt: B > < Pr | Price: M > Cfcp)
                    => < Pr | Price: M > < Pr | AvailQt: B > getCfIntfAQt(Cfcp) .
   rl [getCfIntfAQtf] : getCfIntfAQtf(Cfcpf getCfIntfAQt(Cfcp)) => Cfcpf .
endm
```

Figure B.18: The Maude-based product for cancel interface

HIS_INTF4CANCEL_GNR.maude

```
mod HIS_INTF4CANCEL_GNR is
   inc INTF_GNR .
   protecting RAT .
   sorts HisIntf HisId CustId .
   subsort HisIntf < Intf_NM .
   subsort HisId CustId < Id .

   op HIS : -> Intf_NM .
   op CusId: _ : CustId -> obs_Prop .
   op bill: _ : Rat -> obs_Prop .
   op getCfIntfBill(_, _) : ConfCMP IdL -> ConfCMP .
   op getCfIntfBillf(_) : ConfCMP -> ConfINTF .

   var His : HisId .
   var Cus : CustId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var HisL : IdL .
   var B : Rat .

   rl [getCfIntfBill] : getCfIntfBill(< His | CusId: Cus > < His | bill: B > Cfcp, HisL)
                     => (if belong(His, HisL)
                         then < His | CusId: Cus > < His | bill: B > getCfIntfBill(Cfcp, HisL)
                         else getCfIntfBill(Cfcp, HisL) fi) .
   rl [getCfIntfBillf] : getCfIntfBillf(Cfcpf getCfIntfBill(Cfcp, HisL)) => Cfcpf .
endm
```

Figure B.19: The Maude-based history for cancel interface

The Maude-implementation of the cancel ECA-rule

The Maude-based ECA-driven interaction rule for cancel activity, which is based on the above for cancel interfaces, is depicted in Figure B.20.

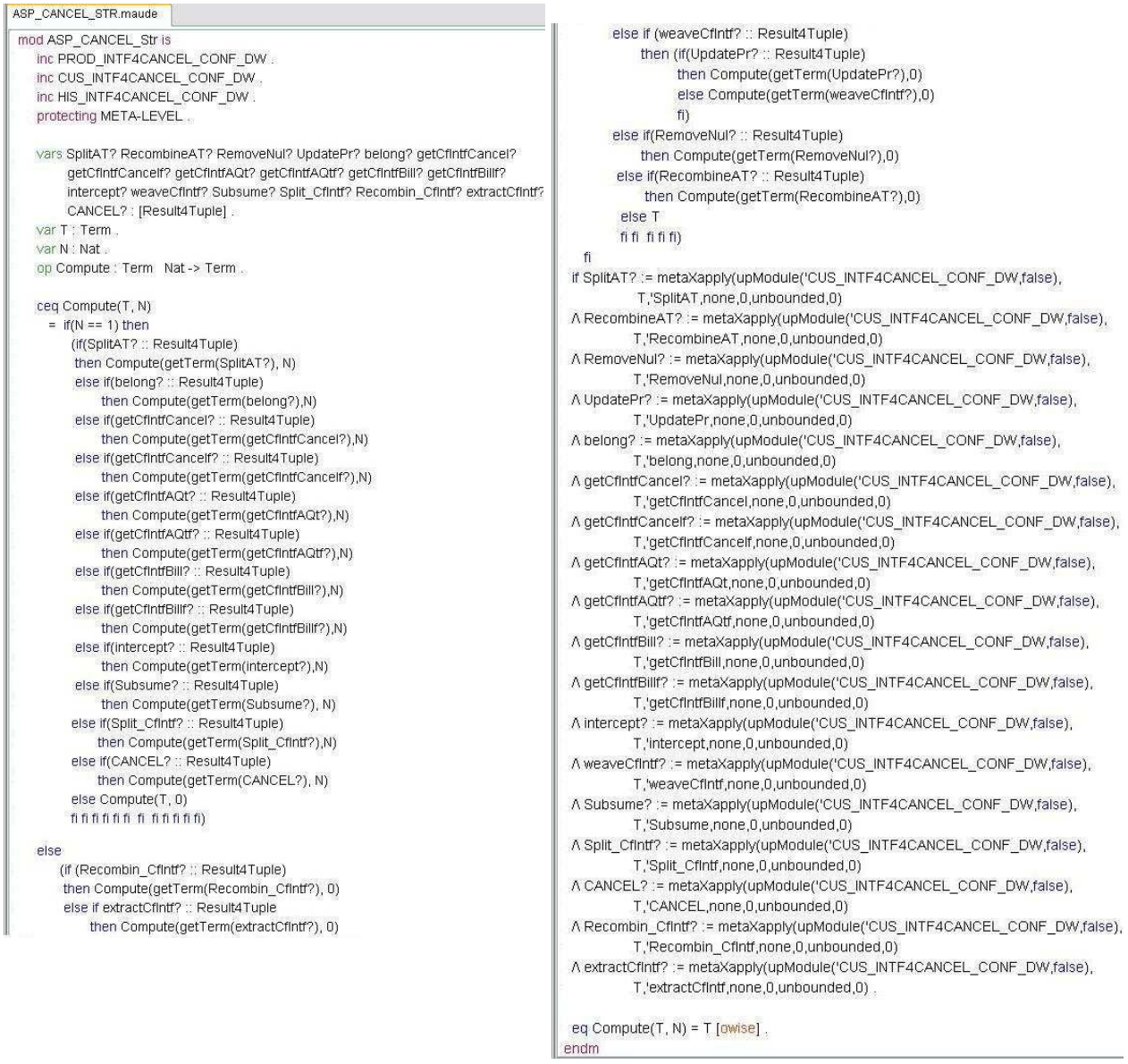
ASP_COORD_CANCEL_GNR.maude

```
mod COORD_CANCEL is
  inc ASP_COORD .
  inc PROD_INTF4CANCEL_GNR .
  inc CUS_INTF4CANCEL_GNR .
  inc HIS_INTF4CANCEL_GNR .
  subsorts PrId CustId HisId < PartnerIds .
  sorts ProfRd Penalty .
  subsorts ProfRd Penalty < Int .

  ops NormalRd SilverRd GoldenRd : -> ProfRd .
  op penalty : -> Penalty .
  op CANCEL : -> Coord_NM .
  op discrate: _ : Int -> Attribute [ctor gather (&)] .
  op Penalty: _ : Int -> Attribute [ctor gather (&)] .
  eq NormalRd = 0 .
  eq SilverRd = 5 .
  eq GoldenRd = 10 .
  eq penalty = 20 .

  vars CS Pr His : PartnerIds .
  var M Q R penal : Int .
  vars B1 B P : Rat .
  rl [CANCEL] : [CANCEL || (CS $ Pr $ His) @ ((discrate: R); (Penalty: penal))] & [CUST | cancelPr(CS, Pr, M)] & [CUST | getBill(CS, B1)]
                & [PROD | < Pr | Price: P >] & [PROD | < Pr | AvailQt: Q >] & [HIS | < His | CusId: CS >] & [HIS | < His | bill: B >]
             => if((B - ((M * P) * ((100 - R) / 100))) > 0)
                then [CANCEL || (CS $ Pr $ His) @ ((discrate: R); (Penalty: penal))] &  [CUST | getBill(CS, (B1 - ((M * P) * ((100 - R - penal) / 100))))]
                     & [PROD | < Pr | Price: P >] & [PROD | UpdatePr(Pr, (- M)) < Pr | AvailQt: Q >] & [HIS | < His | CusId: CS >]
                     & [HIS | < His | bill: (B - ((M * P) * ((100 - R) / 100))) >]
                else [CANCEL || (CS $ Pr $ His) @ ((discrate: R); (Penalty: penal))] & [CUST | getBill(CS, (B * (penal / 100)))]
                     & [PROD | < Pr | Price: P >]  & [PROD | UpdatePr(Pr, (- M)) < Pr | AvailQt: Q >] & [HIS | < His | CusId: CS >] & [HIS | < His | bill: 0 >]
                fi .
endm
```

Figure B.20: The Maude-based cancel coordination

Dynamic weaving of the cancel process as Maude strategy

To control the performance of cancel process, we define the complete Maude reflection strategy in Figure B.21.



```
ASP_CANCEL_STR.maude
mod ASP_CANCEL_Str is
    inc PROD_INTF4CANCEL_CONF_DW .
    inc CUS_INTF4CANCEL_CONF_DW .
    inc HIS_INTF4CANCEL_CONF_DW .
    protecting META-LEVEL .

    vars SplitAT? RecombineAT? RemoveNul? UpdatePr? belong? getCfIntfCancel?
         getCfIntfCancelf? getCfIntfAQt? getCfIntfAQtf? getCfIntfBill? getCfIntfBillf?
         intercept? weaveCfIntf? Subsume? Split_CfIntf? Recombin_CfIntf? extractCfIntf?
         CANCEL? : [Result4Tuple] .
    var T : Term .
    var N : Nat .
    op Compute : Term   Nat -> Term .

    ceq Compute(T, N)
      =  if(N == 1) then
          (if(SplitAT? :: Result4Tuple)
           then Compute(getTerm(SplitAT?), N)
           else if(belong? :: Result4Tuple)
                then Compute(getTerm(belong?),N)
           else if(getCfIntfCancel? :: Result4Tuple)
                then Compute(getTerm(getCfIntfCancel?),N)
           else if(getCfIntfCancelf? :: Result4Tuple)
                then Compute(getTerm(getCfIntfCancelf?),N)
           else if(getCfIntfAQt? :: Result4Tuple)
                then Compute(getTerm(getCfIntfAQt?),N)
           else if(getCfIntfAQtf? :: Result4Tuple)
                then Compute(getTerm(getCfIntfAQtf?),N)
           else if(getCfIntfBill? :: Result4Tuple)
                then Compute(getTerm(getCfIntfBill?),N)
           else if(getCfIntfBillf? :: Result4Tuple)
                then Compute(getTerm(getCfIntfBillf?),N)
           else if(intercept? :: Result4Tuple)
                then Compute(getTerm(intercept?),N)
           else if(Subsume? :: Result4Tuple)
                then Compute(getTerm(Subsume?), N)
          else if(Split_CfIntf? :: Result4Tuple)
               then Compute(getTerm(Split_CfIntf?),N)
          else if(CANCEL? :: Result4Tuple)
               then Compute(getTerm(CANCEL?), N)
          else Compute(T, 0)
          fi fi fi fi fi fi  fi  fi fi fi fi fi)

    else
        (if (Recombin_CfIntf? :: Result4Tuple)
         then Compute(getTerm(Recombin_CfIntf?), 0)
         else if extractCfIntf? :: Result4Tuple
              then Compute(getTerm(extractCfIntf?), 0)
         else if (weaveCfIntf? :: Result4Tuple)
              then (if(UpdatePr? :: Result4Tuple)
                    then Compute(getTerm(UpdatePr?),0)
                    else Compute(getTerm(weaveCfIntf?),0)
                    fi)
         else if(RemoveNul? :: Result4Tuple)
              then Compute(getTerm(RemoveNul?),0)
         else if(RecombineAT? :: Result4Tuple)
              then Compute(getTerm(RecombineAT?),0)
         else T
         fi fi  fi fi fi)
    fi
if SplitAT? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'SplitAT,none,0,unbounded,0)
/\ RecombineAT? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'RecombineAT,none,0,unbounded,0)
/\ RemoveNul? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'RemoveNul,none,0,unbounded,0)
/\ UpdatePr? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'UpdatePr,none,0,unbounded,0)
/\ belong? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'belong,none,0,unbounded,0)
/\ getCfIntfCancel? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'getCfIntfCancel,none,0,unbounded,0)
/\ getCfIntfCancelf? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'getCfIntfCancelf,none,0,unbounded,0)
/\ getCfIntfAQt? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'getCfIntfAQt,none,0,unbounded,0)
/\ getCfIntfAQtf? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'getCfIntfAQtf,none,0,unbounded,0)
/\ getCfIntfBill? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'getCfIntfBill,none,0,unbounded,0)
/\ getCfIntfBillf? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'getCfIntfBillf,none,0,unbounded,0)
/\ intercept? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'intercept,none,0,unbounded,0)
/\ weaveCfIntf? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'weaveCfIntf,none,0,unbounded,0)
/\ Subsume? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'Subsume,none,0,unbounded,0)
/\ Split_CfIntf? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'Split_CfIntf,none,0,unbounded,0)
/\ CANCEL? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'CANCEL,none,0,unbounded,0)
/\ Recombin_CfIntf? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'Recombin_CfIntf,none,0,unbounded,0)
/\ extractCfIntf? := metaXapply(upModule('CUS_INTF4CANCEL_CONF_DW,false),
          T,'extractCfIntf,none,0,unbounded,0) .

 eq Compute(T, N) = T [owise] .
endm
```

Figure B.21: The reflective strategy for cancel activity

B.1.4 The Shipment Activity

The required interfaces for shipment activity

The customer and history components are required in the shipment activity at the component level. Here we depict their corresponding interfaces in Figure B.22 and B.23.

CUS_INTF4SHIP_GNR.maude

```
mod CUS_INTF4SHIP_GNR is
  inc INTF_GNR .
  protecting RAT .
  sorts GB Ask4D CSIntf CustId Shipment .
  subsorts GB Ask4D < Event .
  subsort CSIntf < Intf_NM .
  subsort CustId < Id .

  op CUST : -> Intf_NM .
 ops TakeYself Normal Express : -> Shipment .
  op getBill(_, _) : CustId Rat -> GB .
  op ask4Deliver(_, _) : CustId Shipment -> Ask4D .
  op getCfIntfAsk4D(_, _) : ConfCMP IdL -> ConfCMP .
  op getCfIntfAsk4Df(_) : ConfCMP -> ConfINTF .

  var CS : CustId .
  var Cfcp : ConfCMP .
  var Cfcpf : ConfINTF .
  var CusL : IdL .
  var B : Rat .
  var ship : Shipment .

  rl [getCfIntfAsk4D] : getCfIntfAsk4D(getBill(CS, B) ask4Deliver(CS, ship) Cfcp, CusL)
                     => (if belong(CS, CusL)
                         then getBill(CS, B) ask4Deliver(CS, ship) getCfIntfAsk4D(Cfcp, CusL)
                         else getCfIntfAsk4D(Cfcp, CusL) fi) .
  rl [getCfIntfAsk4Df] : getCfIntfAsk4Df(Cfcpf getCfIntfAsk4D(Cfcp, CusL)) => Cfcpf .
endm
```

Figure B.22: The Maude-based customer for shipment interface

HIS_INTF4SHIP_GNR.maude

```
mod HIS_INTF4SHIP_GNR is
   inc INTF_GNR .
   protecting RAT .
   sorts HisIntf HisId CustId .
   subsort HisIntf < Intf_NM .
   subsort HisId CustId < Id .

   op HIS : -> Intf_NM .
   op CusId: _ : CustId -> obs_Prop .
   op bill: _ : Rat -> obs_Prop .
   op getCfIntfBill(_, _) : ConfCMP IdL -> ConfCMP .
   op getCfIntfBillf(_) : ConfCMP -> ConfINTF .

   var His : HisId .
   var Cus : CustId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var HisL : IdL .
   var B : Rat .

   rl [getCfIntfBill] : getCfIntfBill(< His | CusId: Cus > < His | bill: B > Cfcp, HisL)
                     => (if belong(His, HisL)
                         then < His | CusId: Cus > < His | bill: B > getCfIntfBill(Cfcp, HisL)
                         else getCfIntfBill(Cfcp, HisL) fi) .
   rl [getCfIntfBillf] : getCfIntfBillf(Cfcpf getCfIntfBill(Cfcp, HisL)) => Cfcpf .
endm
```

Figure B.23: The Maude-based history for shipment interface

The Maude-implementation of the shipment ECA-rule

The Maude-based ECA-driven interaction for shipment activity is depicted in Figure B.24.

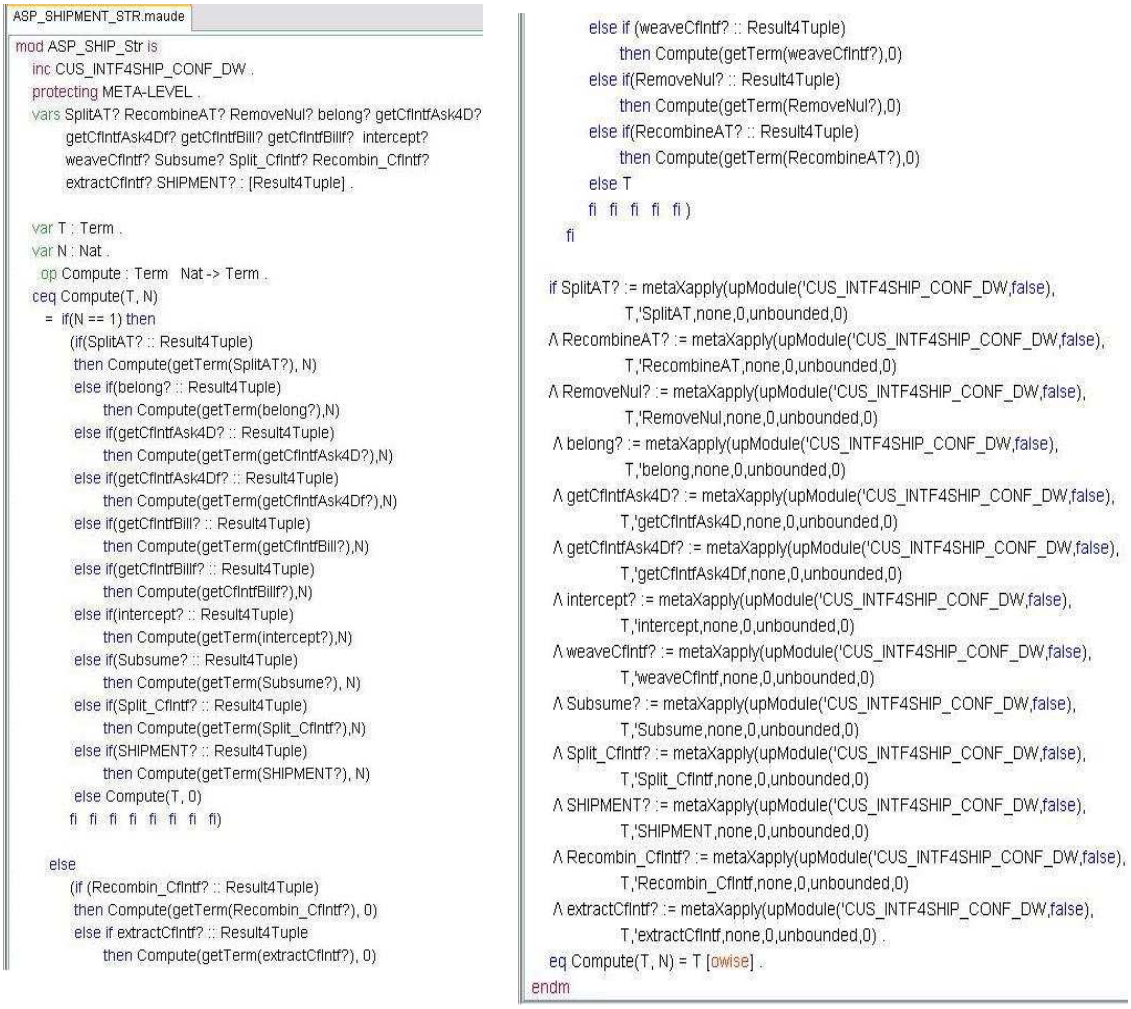
ASP_COORD_SHIPMENT_GNR.maude

```
mod COORD_SHIP is
  inc ASP_COORD .
  inc CUS_INTF4SHIP_GNR .
  inc HIS_INTF4SHIP_GNR .
  subsorts CustId HisId < PartnerIds .
  sort Postage .
  subsort Postage < Int .

  ops TakeYselfPostage NormalPostage ExpressPostage : -> Postage .
  op SHIPMENT : -> Coord_NM .
   op postage: _ : Int -> Attribute [ctor gather (&)] .

  eq TakeYselfPostage = 0 .
  eq NormalPostage = 20 .
  eq ExpressPostage = 40 .

  vars CS Prvd ShipCom His : PartnerIds .
  vasr B B1 : Rat .
  var post : Int .
  var ship : Shipment .

 crl [SHIPMENT] : [SHIPMENT || (CS $ Prvd $ ShipCom) @ postage: post] & [CUST | ask4Deliver(CS, ship)]
                    & [CUST | getBill(CS, B)] & [His | CusId: CS] & [His | bill: B]
                => [SHIPMENT || (CS $ Prvd $ ShipCom) @ postage: post] & [CUST | getBill(CS, (B + post))]
                    & [His | CusId: CS] & [His | bill: B]   if B > 0.
endm
```

Figure B.24: The Maude-based shipment coordination

Dynamic weaving of the shipment process as Maude strategy

In Figure B.25 we depict the complete specific Maude strategy to control the performance of shipment process.



```
ASP_SHIPMENT_STR.maude
mod ASP_SHIP_Str is
  inc CUS_INTF4SHIP_CONF_DW .
  protecting META-LEVEL .
  vars SplitAT? RecombineAT? RemoveNul? belong? getCfIntfAsk4D?
       getCfIntfAsk4Df? getCfIntfBill? getCfIntfBillf?  intercept?
       weaveCfIntf? Subsume? Split_CfIntf? Recombin_CfIntf?
       extractCfIntf? SHIPMENT? : [Result4Tuple] .

  var T : Term .
  var N : Nat .
   op Compute : Term  Nat -> Term .
  ceq Compute(T, N)
    = if(N == 1) then
        (if(SplitAT? :: Result4Tuple)
         then Compute(getTerm(SplitAT?), N)
         else if(belong? :: Result4Tuple)
              then Compute(getTerm(belong?),N)
         else if(getCfIntfAsk4D? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAsk4D?),N)
         else if(getCfIntfAsk4Df? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAsk4Df?),N)
         else if(getCfIntfBill? :: Result4Tuple)
              then Compute(getTerm(getCfIntfBill?),N)
         else if(getCfIntfBillf? :: Result4Tuple)
              then Compute(getCfIntfBillf?),N)
         else if(intercept? :: Result4Tuple)
              then Compute(getTerm(intercept?),N)
         else if(Subsume? :: Result4Tuple)
              then Compute(getTerm(Subsume?), N)
         else if(Split_CfIntf? :: Result4Tuple)
              then Compute(getTerm(Split_CfIntf?),N)
         else if(SHIPMENT? :: Result4Tuple)
              then Compute(getTerm(SHIPMENT?), N)
         else Compute(T, 0)
        fi  fi  fi  fi  fi  fi  fi  fi)

      else
        (if (Recombin_CfIntf? :: Result4Tuple)
         then Compute(getTerm(Recombin_CfIntf?), 0)
         else if extractCfIntf? :: Result4Tuple
              then Compute(getTerm(extractCfIntf?), 0)
         else if (weaveCfIntf? :: Result4Tuple)
              then Compute(getTerm(weaveCfIntf?),0)
         else if(RemoveNul? :: Result4Tuple)
              then Compute(getTerm(RemoveNul?),0)
         else if(RecombineAT? :: Result4Tuple)
              then Compute(getTerm(RecombineAT?),0)
         else T
         fi  fi  fi  fi  fi  fi )
      fi

  if SplitAT? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'SplitAT,none,0,unbounded,0)
  /\ RecombineAT? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'RecombineAT,none,0,unbounded,0)
  /\ RemoveNul? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'RemoveNul,none,0,unbounded,0)
  /\ belong? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'belong,none,0,unbounded,0)
  /\ getCfIntfAsk4D? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'getCfIntfAsk4D,none,0,unbounded,0)
  /\ getCfIntfAsk4Df? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'getCfIntfAsk4Df,none,0,unbounded,0)
  /\ intercept? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'intercept,none,0,unbounded,0)
  /\ weaveCfIntf? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'weaveCfIntf,none,0,unbounded,0)
  /\ Subsume? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'Subsume,none,0,unbounded,0)
  /\ Split_CfIntf? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'Split_CfIntf,none,0,unbounded,0)
  /\ SHIPMENT? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'SHIPMENT,none,0,unbounded,0)
  /\ Recombin_CfIntf? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'Recombin_CfIntf,none,0,unbounded,0)
  /\ extractCfIntf? := metaXapply(upModule('CUS_INTF4SHIP_CONF_DW,false),
                 T,'extractCfIntf,none,0,unbounded,0) .
  eq Compute(T, N) = T [owise] .
endm
```

Figure B.25: The reflective strategy for Shipment activity

B.1.5 The Payment Activity

The required interfaces for payment activity

In payment activity we require customer component, account component and bank card component at the component level. For payment interaction, their corresponding interfaces are required. In Figure B.26 the customer interface for payment is depicted, in Figure B.27 the account interface is depicted, and the bank card interface is depicted in Figure B.28.

CUS_INTF4PAY_GNR.maude

```
mod CUS_INTF4PAY_GNR is
  inc INTF_GNR .
  protecting RAT .
  sorts GB Ask4P SuccP CSIntf CustId .
  subsorts GB Ask4P SuccP < Event .
  subsort CSIntf < Intf_NM .
  subsort CustId < Iid .

  op CUST : -> Intf_NM .
  op getBill(_, _) : CustId Rat -> GB .
  op ask4Pay(_) : CustId -> Ask4P .
  op succPay(_) : CustId  -> SuccP .
  op getCfIntfAsk4P(_, _) : ConfCMP IidL -> ConfCMP .
  op getCfIntfAsk4Pf(_) : ConfCMP -> ConfINTF .

  var CS : CustId .
  var Cfcp : ConfCMP .
  var Cfcpf : ConfINTF .
  var CusL : IidL .
  var B : Rat .

  rl [getCfIntfAsk4P] : getCfIntfAsk4P(getBill(CS, B) ask4Pay(CS) Cfcp, CusL)
                     => (if belong(CS, CusL)
                          then getBill(CS, B) ask4Pay(CS) getCfIntfAsk4P(Cfcp, CusL)
                         else getCfIntfAsk4P(Cfcp, CusL)
                         fi) .
  rl [getCfIntfAsk4Pf] : getCfIntfAsk4Pf(Cfcpf getCfIntfAsk4P(Cfcp, CusL)) => Cfcpf .
endm
```

Figure B.26: The Maude-based customer for payment interface

ACNT_INTF4PAY_GNR.maude

```
mod ACNT_INTF4PAY_GNR is
   inc INTF_GNR .
   protecting RAT .
   sorts TRS ACIntf AcntId .
   subsorts TRS < obs_Msg .
   subsort ACIntf < Intf_NM .
   subsort AcntId < Id .

   op ACNT : -> Intf_NM .
   op Trs(_, _, _) : AcntId AcntId Rat -> TRS .
   op bal: _ : Rat -> obs_Prop [ctor gather (&)] .
   op getCfIntfbal(_, _) : ConfCMP IdL -> ConfCMP .
   op getCfIntfbalf(_) : ConfCMP -> ConfINTF .
   op pendTRS(_) : ConfCMP -> ConfCMP .
   op pendTRSf(_) : ConfCMP -> ConfINTF .
   op backTRS(_, _) : ConfINTF ConfCMP -> ConfCMP .

   vars AC1, AC2 : AcntId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   vars B M : Rat .
   var AcntsL : IdL .

   rl [getCfIntfbal] : getCfIntfbal(< AC | bal: B > Cfcp, AcntsL)
                    => (if belong(AC, AcntsL)
                        then < AC | bal: B > getCfIntfbal(Cfcp, AcntsL)
                        else getCfIntfbal(Cfcp, AcntsL) fi) .
   rl [getCfIntfbalf] : getCfIntfbalf(Cfcpf getCfIntfbal(Cfcp, AcntsL)) => Cfcpf .

   rl [pendTRS] : pendTRS(Trs(AC1, AC2, M) Cfcp) => Trs(AC1, AC2, M) pendTRS(Cfcp) .
   rl [pendTRSf] : pendTRSf(Cfcpf pendTRS(Cfcp)) => Cfcpf .
   rl [backTRSf] : backTRS(Cfcpf, Cfcp) => Cfcpf Cfcp .
endm
```

Figure B.27: The Maude-based account for payment interface

BCARD_INTF4PAY_GNR.maude

```
mod BCARD_INTF4PAY_GNR is
   inc INTF_GNR .
   protecting INT .
   sorts BCardIntf AcntId BCardId .
   subsort BCardIntf < Intf_NM .
   subsort AcntId BCardId < Iid .

   op BCARD : -> Intf_NM .
   op AcntNr: _ : AcntId -> obs_Prop [ctor gather (&)] .
   op getCfIntfBC(_, _) : ConfCMP IidL -> ConfCMP .
   op getCfIntfBCf(_) : ConfCMP -> ConfINTF .

   var BC : BCardId .
   var AC : AcntId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var BCardL : IidL .

   rl [getCfIntfBC] : getCfIntfBC(< BC | AcntNr: AC >  Cfcp, BCardL)
                   => (if belong(BC, BCardL)
                       then < BC | AcntNr: AC > getCfIntfBC(Cfcp, BCardL)
                       else getCfIntfBC(Cfcp, BCardL) fi) .
   rl [getCfIntfBCf] : getCfIntfBCf(Cfcpf getCfIntfBC(Cfcp, BCardL)) => Cfcpf .
endm
```

Figure B.28: The Maude-based bank-card for payment interface

The Maude-implementation of the payment ECA-rule

The complete specific ECA-driven interaction rule for payment activity is depicted in Figure B.29.

ASP_COORD_PAY_GNR.maude

```
mod COORD_PAY is
  inc ASP_COORD .
  inc CUS_INTF4PAY_GNR .
  inc ACNT_INTF4PAY_GNR .
  inc BCARD_INTF4PAY_GNR .
  subsorts CustId BCardId AcntId < PartnerIds .

  op Payment : -> Coord_NM .
  op acnt: _ : AcntId -> Attribute [ctor gather (&)] .

  vars CS BC AC : PartnerIds .
   vars M B B1 : Rat .
   vars AC1 : AcntId .

  crl [Payment] : [Payment || (CS $ BC $ AC) @ acnt: AC1]  & [CUST | ask4Pay(CS)] & [CUST | getBill(CS, M)]
                  & [BCARD | < BC | AcntNr: AC >] & [ACNT | < AC | bal: B >] & [ACNT | < AC1 | bal: B1 >]
               => [Payment || (CS $ BC $ AC) @ acnt: AC1]  & [CUST | succPay(CS)] & [BCARD | < BC | AcntNr: AC >]
                  & [ACNT | Trs(AC, AC1, M) < AC | bal: B > < AC1 | bal: B1 >]   if B >= M .
endm
```

Figure B.29: The Maude-based payment coordination

Dynamic weaving of the payment process as Maude strategy

Figure B.30 depicts the complete specific Maude strategy for payment process.

ASP_PAY_STR.maude

```
mod ASP_PAY_Str is
  inc CUS_INTF4PAY_CONF_DW .
  inc ACNT_INTF4PAY_CONF_DW .
  protecting META-LEVEL .

  vars SplitAT? RecombineAT? RemoveNul? Transfer? pendTRS? pendTRSf?
       backTRS? belong? getCfIntfbal? getCfIntfbalf? getCfIntfAsk4P?
       getCfIntfGetBf? intercept? weaveCfIntf? Subsume? Split_CfIntf?
       Recombin_CfIntf? extractCfIntf? Payment? : [Result4Tuple] .
  var T : Term .
  var N : Nat .
  op Compute : Term  Nat -> Term .

  ceq Compute(T, N)
    = if(N == 1) then
        (if(SplitAT? :: Result4Tuple)
         then Compute(getTerm(SplitAT?), N)
         else if(pendTRS? :: Result4Tuple)
              then Compute(getTerm(pendTRS? ),N)
         else if(pendTRSf? :: Result4Tuple)
              then Compute(getTerm(pendTRSf?),N)
         else if(belong? :: Result4Tuple)
              then Compute(getTerm(belong?),N)
         else if(getCfIntfbal? :: Result4Tuple)
              then Compute(getTerm(getCfIntfbal?), N)
         else if(getCfIntfbalf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfbalf?), N)
         else if(getCfIntfAsp4P? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAsp4P?), N)
         else if(getCfIntfAsk4Pf? :: Result4Tuple)
              then Compute(getTerm(getCfIntfAsk4Pf?), N)
         else if(Subsume? :: Result4Tuple)
              then Compute(getTerm(Subsume?), N)
         else if(Split_CfIntf? :: Result4Tuple)
              then Compute(getTerm(Split_CfIntf?),N)
         else if(Payment? :: Result4Tuple)
              then Compute(getTerm(Payment?), N)
         else Compute(T, 0)
        fi  fi  fi  fi  fi  fi  fi  fi  fi  fi  fi)

    else
        (if (Recombin_CfIntf? :: Result4Tuple)
         then Compute(getTerm(Recombin_CfIntf?), 0)
         else if extractCfIntf? :: Result4Tuple
              then Compute(getTerm(extractCfIntf?), 0)
         else if(intercept? :: Result4Tuple)
              then Compute(getTerm(intercept?),0)
         else if (weaveCfIntf? :: Result4Tuple)
              then (if( Transfer? :: Result4Tuple)
                    then Compute(getTerm( Transfer?),0)
                    else Compute(getTerm(weaveCfIntf?),0)
                    fi)
         else if(backTRS? :: Result4Tuple)
              then Compute(getTerm(backTRS?),0)
         else if(RemoveNul? :: Result4Tuple)
              then Compute(getTerm(RemoveNul?),0)
         else if(RecombineAT? :: Result4Tuple)
              then Compute(getTerm(RecombineAT?),0)
         else T
         fi  fi  fi  fi  fi  fi  fi  fi)
    fi

  if SplitAT? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'SplitAT,none,0,unbounded,0)
  /\ RecombineAT? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'RecombineAT,none,0,unbounded,0)
  /\ RemoveNul? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'RemoveNul,none,0,unbounded,0)
  /\ Transfer? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'Transfer ,none,0,unbounded,0)
  /\ belong? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'belong,none,0,unbounded,0)
  /\ pendTRS? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
          T,'pendTRS,none,0,unbounded,0)
  /\ pendTRSf? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
          T,'pendTRSf,none,0,unbounded,0)
  /\ backTRS? := metaXapply(upModule('ACNT_INTF_CONF_DW,false),
          T,'backTRS,none,0,unbounded,0)
  /\ intercept? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'intercept,none,0,unbounded,0)
  /\ weaveCfIntf? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'weaveCfIntf,none,0,unbounded,0)
  /\ Subsume? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'Subsume,none,0,unbounded,0)
  /\ Split_CfIntf? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'Split_CfIntf,none,0,unbounded,0)
  /\ Recombin_CfIntf? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'Recombin_CfIntf,none,0,unbounded,0)
  /\ extractCfIntf? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'extractCfIntf,none,0,unbounded,0)
  /\ getCfIntfbal? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'getCfIntfbal,none,0,unbounded,0)
  /\ getCfIntfbalf? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'getCfIntfbalf,none,0,unbounded,0)
  /\ getCfIntfGetB? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'getCfIntfGetB,none,0,unbounded,0)
  /\ getCfIntfGetBf? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'getCfIntfGetBf,none,0,unbounded,0)
  /\ Payment? := metaXapply(upModule('CUS_INTF4PAY_CONF_DW,false),
          T,'Payment,none,0,unbounded,0) .
  eq Compute(T, N) = T [owise] .
endm
```

Figure B.30: The reflective strategy for payment activity

B.1.6 The Change-profile Activity

The required interfaces for change-profile activity

In both change-profile and regular change-profile activities, the customer and history components are required at component level. Their corresponding interfaces for both activities are depicted below. In Figure B.31 the specific customer interface is depicted, and in Figure B.32 the history interface is depicted.

CUS_INTF4CHGPF_GNR.maude

```
mod CUS_INTF4CHGPF_GNR is
   inc INTF_GNR .
   protecting RAT .
   sorts  SuccP CSIntf CustId Prof .
   subsort  SuccP < Event .
   subsort CSIntf < Intf_NM .
   subsort CustId < Id .

   op CUST : -> Intf_NM .
   op succPay(_) : CustId  -> SuccP .
   ops Normal Silver Golden : -> Prof .
   op Profile: _ : Prof -> obs_Prop  .
   op getCfIntfProf(_, _) : ConfCMP IdL -> ConfCMP .
   op getCfIntfProff(_) : ConfCMP -> ConfINTF .
   op getCfIntfSuccP(_, _) : ConfCMP IdL -> ConfCMP .
   op getCfIntfSuccPf(_) : ConfCMP -> ConfINTF .

   var CS : CustId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var CusL : IdL .
   var prf : Prof .

   rl [getCfIntfProf] : getCfIntfProf(< CS | Profile: prf > Cfcp, CusL)
                    => (if belong(CS, CusL)
                        then < CS | Profile: prf > getCfIntfProf(Cfcp, CusL)
                        else getCfIntfProf(Cfcp, CusL) fi) .
   rl [getCfIntfProff] : getCfIntfProff(Cfcpf getCfIntfProf(Cfcp, CusL)) => Cfcpf .

   rl [getCfIntfSuccP] : getCfIntfSuccP(succPay(CS) Cfcp, CusL)
                     => (if belong(CS, CusL)
                         then succPay(CS) getCfIntfSuccP(Cfcp, CusL)
                         else getCfIntfSuccP(Cfcp, CusL) fi) .
  rl [getCfIntfSuccPf] : getCfIntfSuccPf(Cfcpf getCfIntfSuccP(Cfcp, CusL)) => Cfcpf .
endm
```

Figure B.31: The Maude-based customer for change profile interface

HIS_INTF4CHGPF_GNR.maude

```
mod HIS_INTF4CHGPF_GNR is
   inc INTF_GNR .
   protecting RAT .
   protecting DATE .
   sorts UPDSP HisIntf HisId CustId GCDAY .
   subsort UPDSP < obs_Msg .
   subsort HisIntf < Intf_NM .
   subsort HisId CustId < Iid .
   subsort GCDAY < Event .

   op HIS : -> Intf_NM .
   op UpdateSp(_, _) : CustId Rat -> UPDSP [ctor] .
   op CusId: _ : CustId -> obs_Prop .
   ops Spending: _ : Rat -> obs_Prop .
   op bill: _ : Rat -> obs_Prop .
   op LastUpdateSP: _ : Date -> obs_Prop .
   op getCurrentDay(_, _) : HisId Date -> GCDAY [ctor] .
   op getCfIntfSpd(_, _) : ConfCMP IidL -> ConfCMP .
   op getCfIntfSpdf(_) : ConfCMP -> ConfINTF .

   var His : HisId .
   var Cus : CustId .
   var Cfcp : ConfCMP .
   var Cfcpf : ConfINTF .
   var HisL : IidL .
   vars date today : Date .
   vars M B : Rat .
   rl [getCfIntfSpd] : getCfIntfSpd(getCurrentDay(His, today) < His | CusId: Cus > < His | bill: B >
                       < His | LastUpdateSP: date > < His | Spending: M > Cfcp, HisL)
                    => (if belong(His, HisL)
                        then getCurrentDay(His, today) < His | CusId: Cus > < His | bill: B >
                             < His | LastUpdateSP: date >  < His | Spending: M > getCfIntfSpd(Cfcp, HisL)
                        else getCfIntfSpd(Cfcp, HisL) fi) .
  rl [getCfIntfSpdf] : getCfIntfSpdf(Cfcpf getCfIntfSpd(Cfcp, HisL)) => Cfcpf .
endm
```

Figure B.32: The Maude-based history for change profile interface

The Maude-implementation of the change-profile ECA-rule

The ECA-driven interaction rules change-profile and regular change-profile are depicted in Figure B.33.

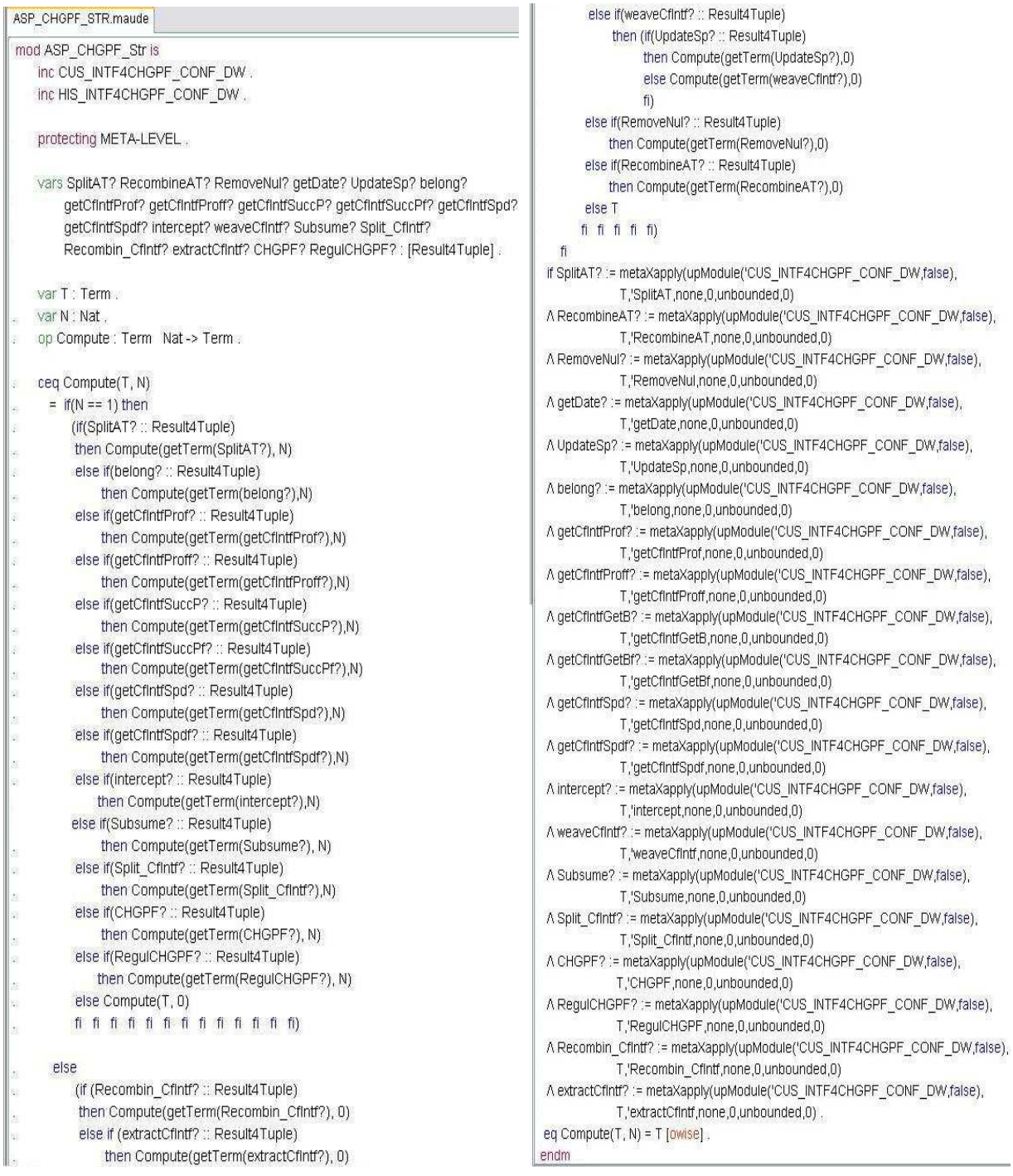
ASP_COORD_CHGPF_GNR.maude

```
mod COORD_CHGPF is
    inc ASP_COORD .
    inc CUS_INTF4CHGPF_GNR .
    inc HIS_INTF4CHGPF_GNR .
    subsorts  CustId HisId < PartnerIds .

    op CHGPF : -> Coord_NM .
    op RegulCHGPF : -> Coord_NM .
    op interval: _ : Int -> Attribute [ctor gather (&)] .
    op currentday: _ : Date -> Attribute [ctor gather (&)] .
    ops MinSilver MaxSilver INTERVAL : -> Int .

    eq MinSilver = 1000 .
    eq MaxSilver = 3000 .
    eq INTERVAL = 3 .

    vars CS His: PartnerIds .
    vars B B1 S : Rat .
    var Mt : Int .
    var date today : Date .
    var prf : Prof .
    rl [CHGPF] : [CHGPF || (CS $ His) @ currentday: today] & [CUST | < CS | Profile: prf >] & [CUST | succPay(CS)]
                 & [HIS | < His | CusId: CS >] & [HIS | < His | bill: B >] & [HIS | < His | LastUpdateSP: date >] &  [HIS | < His | Spending: S >]

              => if (prf == Normal and (S + B) > MinSilver)
                 then [CHGPF || (CS $ His) @ currentday: today] & [CUST | < CS | Profile: Silver >]
                      & [HIS | UpdateSp(CS, B) < His | CusId: CS > < His | Spending: S >]
                      & [HIS | < His | bill: 0 > < His | LastUpdateSP: today >]
                 else if (prf == Silver and (S + B) > MaxSilver)
                      then [CHGPF || (CS $ His) @ currentday: today] & [CUST | < CS | Profile: Golden >]
                           & [HIS | UpdateSp(CS, B) < His | CusId: CS > < His | Spending: S >]
                           & [HIS | < His | bill: 0 > < His | LastUpdateSP: today >]
                      else if (B > 0)
                           then [CHGPF || (CS $ His) @ currentday: today] & [CUST | < CS | Profile: prf >]
                                & [HIS | UpdateSp(CS, B) < His | CusId: CS > < His | Spending: S >]
                                & [HIS | < His | bill: 0 > < His | LastUpdateSP: today >]
                           else [CHGPF || (CS $ His) @ currentday: today] & [CUST | < CS | Profile: prf >] & [HIS | < His | bill: 0 >]
                                &  < His | CusId: CS >] & [HIS | < His | LastUpdateSP: date >]  &  [HIS | < His | Spending: S >]
                 fi    fi    fi .

   rl [RegulCHGPF] : [RegulCHGPF || (CS $ His) @ interval: Mt] & [HIS | getCurrentDay(His, today)] & [CUST | < CS | Profile: prf >]
                     & [HIS | < His | CusId: CS >] & [HIS | < His | LastUpdateSP: date >] &  [HIS | < His | Spending: S >]

                  => if(GreaterM(today, date, Mt) == true)
                     then (if((prf == Golden) and (S < 3100))
                           then [RegulCHGPF || (CS $ His) @ interval: Mt] & [CUST | < CS | Profile: Silver >]
                                & [HIS | < His | CusId: CS >] & [HIS | < His | LastUpdateSP: today >] &  [HIS | < His | Spending: (S - 100) >]
                           else if((prf == Silver) and (S < 1100))
                                then [RegulCHGPF || (CS $ His) @ interval: Mt] & [CUST | < CS | Profile: Normal >]
                                     & [HIS | < His | CusId: CS >] & [HIS | < His | LastUpdateSP: today >] &  [HIS | < His | Spending: (S - 100) >]
                                else if(S > 100)
                                     then [RegulCHGPF || (CS $ His) @ interval: Mt] & [CUST | < CS | Profile: prf >] & [HIS | < His | CusId: CS >]
                                          & [HIS | < His | LastUpdateSP: today >] & [HIS | < His | Spending: (S - 100) >]
                                     else [RegulCHGPF || (CS $ His) @ interval: Mt] & [CUST | < CS | Profile: prf >] & [HIS | < His | CusId: CS >]
                                          & [HIS | < His | LastUpdateSP: today >] & [HIS | < His | Spending: 0 >]
                           fi    fi    fi)
                     else [RegulCHGPF || (CS $ His) @ interval: Mt] & [CUST | < CS | Profile: prf >] & [HIS | < His | CusId: CS >]
                          & [HIS | < His | LastUpdateSP: date >] &  [HIS | < His | Spending: S >]
                     fi .
endm
```

Figure B.33: The Maude-based change profile coordination

Dynamic weaving of the change-profile process as Maude strategy

The complete specific strategy to control the performance of changing profile processes are shown in Figure B.34.



```
ASP_CHGPF_STR.maude

mod ASP_CHGPF_Str is
    inc CUS_INTF4CHGPF_CONF_DW .
    inc HIS_INTF4CHGPF_CONF_DW .

    protecting META-LEVEL .

    vars SplitAT? RecombineAT? RemoveNul? getDate? UpdateSp? belong?
         getCfIntfProf? getCfIntfProff? getCfIntfSuccP? getCfIntfSuccPf? getCfIntfSpd?
         getCfIntfSpdf? intercept? weaveCfIntf? Subsume? Split_CfIntf?
         Recombin_CfIntf? extractCfIntf? CHGPF? RegulCHGPF? : [Result4Tuple] .

    var T : Term .
    var N : Nat .
    op Compute : Term  Nat -> Term .

    ceq Compute(T, N)
      = if(N == 1) then
          (if(SplitAT? :: Result4Tuple)
           then Compute(getTerm(SplitAT?), N)
           else if(belong? :: Result4Tuple)
                then Compute(getTerm(belong?),N)
           else if(getCfIntfProf? :: Result4Tuple)
                then Compute(getTerm(getCfIntfProf?),N)
           else if(getCfIntfProff? :: Result4Tuple)
                then Compute(getTerm(getCfIntfProff?),N)
           else if(getCfIntfSuccP? :: Result4Tuple)
                then Compute(getTerm(getCfIntfSuccP?),N)
           else if(getCfIntfSuccPf? :: Result4Tuple)
                then Compute(getTerm(getCfIntfSuccPf?),N)
           else if(getCfIntfSpd? :: Result4Tuple)
                then Compute(getTerm(getCfIntfSpd?),N)
           else if(getCfIntfSpdf? :: Result4Tuple)
                then Compute(getTerm(getCfIntfSpdf?),N)
           else if(intercept? :: Result4Tuple)
               then Compute(getTerm(intercept?),N)
          else if(Subsume? :: Result4Tuple)
                then Compute(getTerm(Subsume?), N)
           else if(Split_CfIntf? :: Result4Tuple)
                then Compute(getTerm(Split_CfIntf?),N)
           else if(CHGPF? :: Result4Tuple)
                then Compute(getTerm(CHGPF?), N)
           else if(RegulCHGPF? :: Result4Tuple)
                then Compute(getTerm(RegulCHGPF?), N)
           else Compute(T, 0)
           fi  fi  fi  fi  fi  fi  fi  fi  fi  fi  fi  fi  fi  fi)

      else
          (if (Recombin_CfIntf? :: Result4Tuple)
           then Compute(getTerm(Recombin_CfIntf?), 0)
           else if (extractCfIntf? :: Result4Tuple)
                then Compute(getTerm(extractCfIntf?), 0)
           else if(weaveCfIntf? :: Result4Tuple)
                then (if(UpdateSp? :: Result4Tuple)
                      then Compute(getTerm(UpdateSp?),0)
                      else Compute(getTerm(weaveCfIntf?),0)
                      fi)
           else if(RemoveNul? :: Result4Tuple)
                then Compute(getTerm(RemoveNul?),0)
           else if(RecombineAT? :: Result4Tuple)
                then Compute(getTerm(RecombineAT?),0)
           else T
          fi  fi  fi  fi  fi)
      fi
    if SplitAT? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'SplitAT,none,0,unbounded,0)
    /\ RecombineAT? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'RecombineAT,none,0,unbounded,0)
    /\ RemoveNul? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'RemoveNul,none,0,unbounded,0)
    /\ getDate? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'getDate,none,0,unbounded,0)
    /\ UpdateSp? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'UpdateSp,none,0,unbounded,0)
    /\ belong? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'belong,none,0,unbounded,0)
    /\ getCfIntfProf? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'getCfIntfProf,none,0,unbounded,0)
    /\ getCfIntfProff? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'getCfIntfProff,none,0,unbounded,0)
    /\ getCfIntfGetB? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'getCfIntfGetB,none,0,unbounded,0)
    /\ getCfIntfGetBf? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'getCfIntfGetBf,none,0,unbounded,0)
    /\ getCfIntfSpd? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'getCfIntfSpd,none,0,unbounded,0)
    /\ getCfIntfSpdf? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'getCfIntfSpdf,none,0,unbounded,0)
    /\ intercept? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'intercept,none,0,unbounded,0)
    /\ weaveCfIntf? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'weaveCfIntf,none,0,unbounded,0)
    /\ Subsume? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'Subsume,none,0,unbounded,0)
    /\ Split_CfIntf? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'Split_CfIntf,none,0,unbounded,0)
    /\ CHGPF? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'CHGPF,none,0,unbounded,0)
    /\ RegulCHGPF? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'RegulCHGPF,none,0,unbounded,0)
    /\ Recombin_CfIntf? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'Recombin_CfIntf,none,0,unbounded,0)
    /\ extractCfIntf? := metaXapply(upModule('CUS_INTF4CHGPF_CONF_DW,false),
                   T,'extractCfIntf,none,0,unbounded,0) .
    eq Compute(T, N) = T [owise] .
endm
```

Figure B.34: The reflective strategy for changing profile activity

Appendix C

Introduction to Rewriting Logic and Maude

As noted in the introduction, the approach adopted in the present research is underpinned by rewriting logic and the Maude language [27]. To make this thesis self-contained and coherent, this complete chapter is thus devoted to introducing all the required principles and mechanisms related to this language in a sequential manner. That is, the main features of the Maude language are first outlined, then, since stateless data are described in Maude as *functional modules*, this functional level is reviewed and illustrated. In order to situate the object-oriented applications of Maude, all of the ingredients underlying its *system modules* are next introduced, followed by an explanation of how object-oriented applications can be directly and nicely specified as object modules, with classes, sub-classes, methods and effects of methods as rules. Finally, there is an account of ways to control the rewriting rules using reflection in general, detailing how internal strategies can be specified in Maude.

C.1 Maude: Main Features

Maude is a high-level language and a high-performance system supporting both functional and object-oriented specifications and programming for a wide range of applications. Maude has been influenced in important ways by the OBJ3 functional language [48]. In particular, Maude's underlying equational logic sublanguage, namely, membership equational logic, extends OBJ3's order-sorted equational logic [47]. The main features and characteristics of Maude can be summarised as follows:

Rewriting logic-based semantics. Rewriting logic [71] is logic of concurrent changes that can deal with states and with highly nondeterministic concurrent computations. This makes it particularly well suited to express in a declarative way concurrent and state-changing aspects of systems. Maude programs are theories, and rewriting logic deduction corresponds exactly to concurrent computation. Moreover, rewriting logic is a flexible and general semantic framework for a wide range of languages. At the logical level, Maude allows executable specifications, rapid prototyping, and efficient parallel and distributed executions [29, 72].

Reflection. Rewriting logic and Maude are reflective [25, 29]. That is, they are able to express their own metalevel at the object level. The design of Maude capitalises on this feature to support a novel style of metaprogramming, which includes both user-definable module operations and declarative strategies to guide the deduction process, thus offering powerful module-combining and module-transforming operations that surpass those of traditional parameterised programming. This can greatly advance software reusability and adaptability. The Maude strategies for controlling the rewriting process are defined by rewrite rules at the metalevel and can be reasoned about inside the logic. Therefore, instead of having a "Logic + Control" introduction of extra-logical features, Maude has "*Control* ⊑ *Logic*".

Maude modules are rewriting theories, while computation with such modules corresponds to efficient deduction by rewriting [35]. As just emphasised, there are three kinds of modules in Maude: the functional module (**fmod**), the system module (**mod**) and the object-oriented module (**omod**). Figure C.1 show Maude divisions and there modules.

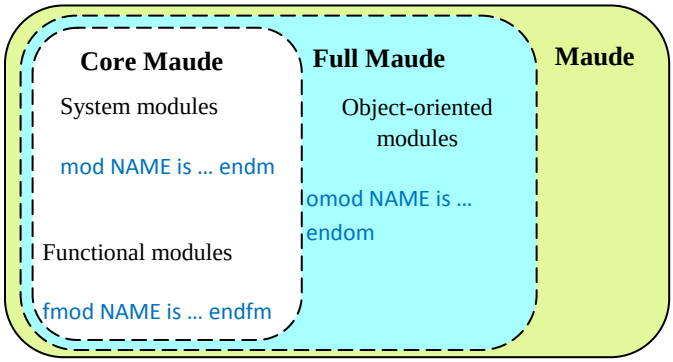


Figure C.1: Maude divisions

Maude's functional modules are theories in membership equational logic, which extends order-sorted equational logic and supports sorts, subsort relations, operator overloading, definition of partial functions with equationally defined domains, and error specification. They are assumed to be Church-Rosser and terminating [36]. Membership equational logic is sublogic of rewriting logic [26]. The Church-Rosser property means that the rewriting or the reduction of a given term should lead to one final term, also called the *normal form* of that term. The termination property ensures that the rewriting process is *not infinite*; thus there always exists a normal form for any given term.

Maude's system modules are rewrite theories, in which the local transition rules in a concurrent system represent the inference rules in a logical system, instead of equations as oriented rewrite rules. In rewrite logic, the rewrite rules need not be terminating and Church-Rosser. In other words, Maude's system modules permit the specifying of reactive long-lifespan systems, which are in continuous interaction with their environment. Information systems such as e-commerce and e-government applications are typical reactive systems and it is thus adequate to specify them as system modules.

Object-oriented modules allow the modelling of an object-oriented system in an explicit and detailed manner. In particular, Maude object-oriented modules permit the specification of classes, subclasses, methods and the effects of methods as explicit rules. For efficiency purposes, object-oriented modules can be automatically translated into general-purpose system modules.

In addition, Maude supports rewriting modulo-equational theories such as associativity, commutativity and identity. Therefore, the user may not only have infinite chains of rewriting, but may also have highly divergent rewriting paths, which could never cross each other by further rewriting. Hence, good ways of controlling the rewriting inference process are needed. Using reflection, it can be controlled with great flexibility in Maude by means of internal strategies. This chapter explains and illustrates with examples all these Maude concepts.

C.2 Functional Modules

Functional modules define data types and operations on them by means of equational theories. Computation in a functional module is accomplished by using the equations as rewrite rules. That is, each rewriting step is a step of replacement of equals by equals, until a canonical form is found. For this reason, the equations in the functional module must satisfy the additional requirements of being Church-Rosser, terminating and sort decreasing.

The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic called membership equational logic [73]. In addition to supporting sorts, subsort relations and overloading of function symbols, functional modules also support membership axioms, a generalisation of sort constraints in which a term is asserted to have a certain sort if a condition consisting of a conjunction of equations and unconditional

membership tests is satisfied. Such membership axioms can be used to define partial functions, which become defined when their arguments satisfy certain equational and membership conditions. The following Maude functional modules, NAT and NAT-REVERSE, illustrate these ideas. It should be noted that Maude supports mixfix user-definable syntax (for operation).

```
1. fmod NAT is
2. sorts NzNat Nat .
3. subsort NzNat < Nat .
4. op 0 : → Nat [ctor] .
5. op s _ : Nat → NzNat .
6. op _ + _ : Nat Nat → Nat [assoc comm].
7. var N : Nat .
8. var M : Nat .
9. cmb N / M : NzNat if (N ≠ 0) .
10. eq N + 0 = N .
11. eq s(N) + M = s(N + M) .
12. endfm
```

```
1. fmod NAT-REVERSE is
2. protecting Nat .
3. sort Tree .
4. subsort Nat < Tree .
5. op nil : → Tree [ctor] .
6. op _ ^ _ : Tree Tree→Tree [assoc comm id:nil].
7. op rev( _ ) : Tree → Tree .
8. var N : Nat .
9. var T : Tree .
10. var T' : Tree .
11. eq rev(N) = N .
12. eq rev(T ^ (T'))=rev(T) ^ rev(T') .
13. endfm
```

The modules are introduced with the functional module syntax *fmod ... endfm* and are named NAT and NAT-REVERSE respectively. The statement *protecting NAT* imports module NAT as a submodule of the NAT-REVERSE module. This asserts that the natural numbers are not modified in the sense that no new data of sort Nat is added and that different numbers are not identified by the new equations declared in NAT-REVERSE.

Sort and subsort: The sort and subsort relations of this module are introduced by a sort and subsort declarations by the keywords *sort(s)* and *subsort(s)*. Sorts are used to classify data. A subsort relation between two sorts is interpreted as a set-theoretic inclusion, meaning that the data of the subsort is included in that of the supersort. For example, in the NAT module, the subsort declaration subsort NzNat < Nat declares that every nonzero natural number is a natural number, which means that the set of natural numbers (*Nat)* contains the nonzero natural number set (*NzNat*).

It is important to mention that Maude has *no line numbers*. They have been added in this thesis only to enhance clarity and facilitate the explanation of any 'line' code.

Operators and their attributes: In general, an operator can be declared with the keyword *op* followed by the name of the operator, then a colon, then the list of sorts for its arguments, followed by an arrow (→), then the sort of its result. The operator can have attributes such as

assoc, *comm* and *id*, which indicate some equational axioms satisfied by the operator and used for term matching. The operator attribute *assoc* means associativity, that is, it makes no difference whether parentheses are left- or right-associated in a concatenation expression, while *comm* is commutativity, which means that order does not matter, e.g. "*0 + s 0 + s s s 0*" is the same as "*s0 + 0 + s s s 0*". Identity is declared with the keyword *id:* followed by the nil constant which fulfils that identity property. All such attributes are declared within a single pair of enclosing square brackets after the sort of the result and before the ending period. For example, in the NAT-REVERSE module, the tree concatenation operator declaration takes the form:

Op _ ^ _ : Tree Tree → Tree [assoc comm id: nil],

where the operator attributes *assoc* and *comm* imply that the tree leaves can be swapped. The operator attribute *id: nil* means that the tree can be an empty tree. Maude also supports a special type of operator attribute called a constructor ([*ctor*]), which is a fundamental operation that defines the basics of the algebra. For example, the constant natural number 0 in the NAT module is declared as a constructor [*ctor*]. Note that for these two modules, initial algebras are just what would be expected: the NAT module specifies the natural numbers, with *zero*, *successor* and addition (+), while the NAT-REVERSE module specifies the binary trees with natural numbers on their leaves, with tree reversal and binary tree constructor operators. In this module, the natural numbers are thus viewed as the subset of trees consisting of a single node.

Membership and variables: Membership refers to the conditions on certain terms being members of given sorts. A variable is declared a member of a sort using the colon, which one can think of a symbol for "is a member of". Thus, the declaration *var N: Nat* is the same as saying "the variable N is a member of the sort Nat". The NAT module contains the declarations:

```
var N : Nat .
var M : NzNat .
cmb N / M : NzNat if (N ≠ 0) .
```

The membership axioms deal with membership and are declared with the keywords *cmb* (for conditional membership) or *mb*. For the above conditional membership axiom there is an equivalent membership axiom:

```
var N : NzNat .
var M : NzNat .
mb N / M : NzNat .
```

(Conditional) equations: These define the semantics of operations. Each equation is introduced by the keyword *eq*—or *ceq* for conditional equations—and has variables of appropriate sorts previously declared with *var* declarations. The equations are then used from left to right by the Maude engine to simplify any expression to its canonical form, that is, to evaluate any expression to its corresponding value. All variables on the right-hand side of an equation should also appear in the corresponding left-hand side. For conditional equations or conditional membership axioms, the conditions should involve only variables appearing in the corresponding membership predicate – that is, in the corresponding left-hand side.

As illustrated by a few sample evaluations and their results, expressions formed with the operators declared in the module can be evaluated with the *reduce* command (*red* in abbreviated form). To compute with such modules, one performs equational simplification by using the equations from left to right until no more simplifications are possible. Note that this can be done concurrently, i.e. by applying several equations at once. Figure C.2 shows the reduction of the example tree.

```
Maude> red in NAT: s s s 0 + s 0.
result NzNat: s s s s 0
Maude> red in NAT-REVERSE: rev ((s s s 0 + s 0) ^ ((s s 0 + 0) ^ 0)).
result Tree: 0 ^ s s 0 ^ s s s s 0
```

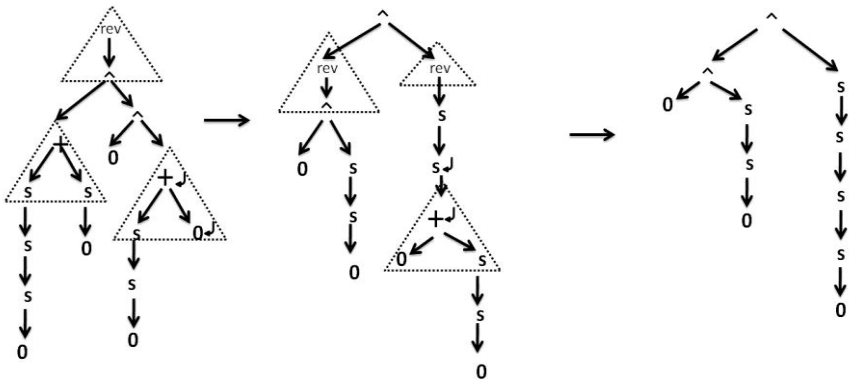


Figure C.2: Tree reduction process

C.3 System and Object-Oriented Modules

In functional modules, the rewrite rules are equations, which must satisfy the additional requirements of being *Church-Rosser*, *terminating* and *sort decreasing*. That is, in such modules, each step of rewriting is a step of replacement of equals by equals, until a canonical form is found. In general, however, a set of rewrite rules need not be terminating and Church-Rosser. That is, it is possible not only to have infinite chains of rewriting, but also to have highly divergent rewriting paths which could never cross each other by further rewriting.

The passage from functional modules to system modules involves a fundamental change in perspective, so that basic notations that previously had a familiar interpretation in functional terms have now to be reinterpreted in a different way. As mentioned already, in this new interpretation, a term *t* is no longer understood as a functional expression, but as a *structured state* of a system, where the structure of the state is given by the operators that happen to appear in the term and by the structural axioms that they enjoy. The algebraic structure of the state—as a multiset, binary tree or whatever—is precisely what makes the state distributed, i.e. coincides with its distributed structure and makes concurrency possible. In the same way, a rewrite rule $t \rightarrow t'$ is no longer seen as functional evaluation by equational deduction, but as a local *state transition*, stating that if a portion of a system's state exhibits the pattern described by *t*, then that portion of the system can change to the corresponding instance of t'. The states are algebraic and therefore the distributed structure makes it possible for many rewritings to occur concurrently, i.e. rewritings are local transitions of a distributed state that happen independently of each other. This discussion can be summarised through the following three correspondences: the usual algebraic term corresponds to the reactive system state; usable rewriting steps correspond to the system (transitions); and the static algebraic structure corresponds to the distributed structure of a reactive system.

State ↔ Term

Transition ↔ Rewriting

Distributed Structure ↔ Algebraic Structure

C.3.1 Maude System Modules

A rewrite theory can be presented as a 4-tuple $R = (\sum, E, L, M)$ where $(\sum, E)$ is a theory in membership equational logic that specifies states of a reactive system as a distributed structure, L is a set of labels for the rules and M is a set of labelled rewrite rules axiomatising the local state transitions of the system with either the unconditional form $rl : [t] \rightarrow [t']$ or the conditional form $crl : [t] \rightarrow [t']$ if C, with C referring the conditions to be observed to apply the rule.

Rewriting logic is a computational logic to specify concurrent systems having states and evolving by means of transitions. The signature of a rewrite theory describes a particular structure for the states of a system, while the rewrite rules denote which elementary local transitions are possible in the distributed state. The inference rules of deduction for rewriting logic encompass the following:

1. Reflexivity: the possibility of having idle transitions.

2. Congruence: a general form of sideways parallelism.

3. Replacement: combines an atomic transition at the top using a rule with nested concurrency in the substitution.

4. Transitivity: sequential composition.

They allow users to infer all the complex concurrent state changes that a system may exhibit, given a set of rewrite rules which describe its elementary local changes.

The most general Maude modules are system modules. They specify the initial module of a rewrite theory $(\sum, E, L, M)$, in which the signature $\sum$ is given by the sorts, subsort relations, and operator declarations, and E is a set of equations that is assumed to be decomposed as a union $E = A \cup E'$, with A being a set of axioms to rewrite modules among those supported by Maude and E' a set of Church-Rosser and termination equations modulo A. These initial modules capture the intuitive idea of *rewrite systems* in the sense that they are transition systems whose states are equivalence classes $[t]$ of ground terms modulo the equations E, and whose transitions are proofs $\alpha : [t] \rightarrow [t']$ in rewriting logic, that is, concurrent rewriting computations in the system described by the rules in R. Such proofs are equated modulo a

natural notion of proof equivalence that computationally corresponds to the 'true concurrency' of the computations.

Example: For a concrete illustration of such Maude system modules, consider a simple machine for buying underground tickets. The behaviour of this ticket-machine can be described as follows:

There are two kinds of ticket, shorter and longer, referring respectively to short and long distances travelled. Further, assume that this machine accepts US dollars ($) and quarters (q). Shorter-distance tickets (t_1) cost two quarters, whereas longer ones (t_2) cost three quarters.

One can buy tickets with quarters or with dollars. With a dollar, a user can buy a ticket of type t_1 by pushing button $b{-}t_1$ and receives two quarters in change. Alternatively, if he pushes button $b{-}t_2$, he gets a t_2 ticket and one quarter in change. The machine can also be used to change a dollar into four quarters by pushing a button called *change*.

To specify this machine with a Maude system module, the internal structure of the machine state is required. This state can be regarded as an arbitrarily ordered multiset of money (dollars and quarters) and tickets (t_1 and t_2). In other words, money and tickets co-exist within the machine. For the sake of simplicity, the empty binary multiset denoted by _ _ can be chosen to capture such a multiset.

Having set these defining features, it is clear that to each of the three button corresponds a given rewrite rule which reflect its functioning behaviour. For instance, the rule governing the shorter ticket button accepts one dollar, then delivers the ticket t_1 and the change of two quarters. That is, the left-hand side of this rule corresponds to $, whereas its right-hand side corresponds to the expected multiset t_1 *q q*. Note that two quarters can also be directly used to buy this shorter ticket. To distinguish this variant from the previous one, *b´* is used instead of *b* in the rule name, which is now $b'{-}t_1$. All in all; the five rules can be straightforwardly specified as follows:

Rule [b $-t_1$] : $ $\Rightarrow t_1$ q q.
Rule [b $-t_2$] : $ $\Rightarrow t_2$ q.
Rule [change] : $ $\Rightarrow$ q q q q.
Rule [b´$-t_1$] : q q $\Rightarrow t_1$.

Rule [b′−t_2] : q q q ⇒t_2.

For the complete Maude system module, a name is required for the internal structure of the machine; let it be called MACHINE-STATE. As has just been discussed, this machine state includes the TICKET-MONEY multiset sort, which is composed of four constants: *$*, *q*, t_1 and t_2 . Also, *nil* is defined as the empty multiset for the machine-state (nothing is in the machine). The corresponding Maude module then takes the following complete machine specification:

1. mod TICKET-MACHINE is
2. sorts MACHINE-STATE .
3. subsorts TICKET-MONEY < MACHINE-STATE .
4. ops $, q, t_1 , t_2 : → TICKET-MONEY.
5. op nil : → MACHINE-STATE .
6. op _ _ : MACHINE-STATE MACHINE-STATE →MACHINE-STATE [assoc comm id:nil].
7. rl [b −t_1] : $ ⇒t_1 q q .
8. rl [b −t_2] : $ ⇒t_2 q .
9. rl [change] : $ ⇒ q q q q .
10. rl [b′−t_1] : q q ⇒t_1 .
11. rl [b′−t_2] : q q q ⇒t_2 .
12. endm

As mentioned above, in the rewrite theory (∑, *E, L, M*) the set *E* of equations is assumed to be decomposed as a union *E* = *A* U *E′*, where A is a set of axioms to rewrite modules among those supported by Maude and *E′* is a set of Church-Rosser and terminating equations modulo A. In the above example, A consists of the associativity, commutativity and identity axioms, while *E′* is empty. The label set *L* contains the labels of the five rules, each starting with the keyword *rl*. One can also define conditional rules, using instead the keyword *crl*.

Assume that the machine initially contains four quarters and two dollars and that the user wants one shorter ticket and three longer ones. In this case, the concurrent application of the five rules results in the following rewriting process.

$$qqqq\$\$ \xrightarrow[b\text{-}\,t_2]{b'\text{-}\,t_1} qq\,t_1t_2q\$ \xrightarrow[b\text{-}\,t_2]{b'\text{-}\,t_2} t_2t_1t_2t_2q$$

Figure C.3: Rewriting process

Using reflection, the rewriting inference process can be controlled with great flexibility in Maude by means of *strategies*. These are defined by rewrite rules at the metalevel (more details of *strategies* are given in a later subsection). However, the Maude interpreter provides a *default strategy* for executing expressions in system modules. The default strategy applies the rules in a top-down fair way, and is provided by the rewrite command, with keyword *rewrite*, abbreviated as *rew*. Since the equations *E* are assumed in a system module to be decomposed as a union $E = A \cup E'$, where *A* is a set of axioms to rewrite modules and E' is a set of Church-Rosser and terminating equations modulo *A*, they are always reduced to canonical form using E' before applying any rule in R. More specifically, before the application of each rewrite rule, the expression is simplified to its canonical form using the equations E' modulo *A*, then a rule is applied to this simplified expression modulo the axioms *A* according to the default strategy.

C.3.2 Object-Oriented Module

To present a logical theory of concurrent objects based on rewriting logic deduction modulo *ACI* (*associativity, commutativity* and *identity*), the key idea is to conceptualise the distributed state of a concurrent object-oriented system—called a configuration—as a multiset of object states and message instances. This multiset evolves by concurrent rewriting modulo associativity, commutativity and identity, using rules that describe the effects of communication events between objects and messages. Intuitively, messages can be thought of as *travelling* to come into contact with the objects to which they are sent, and then causing *communication events* by application of rewrite rules. Therefore, concurrent object-oriented computation can be viewed as deduction in rewriting logic; in this way, the configurations *S* that are reachable from a given initial configuration s_0 are exactly those such that the sequence $s_0 \rightarrow s_1$ is provable in rewriting logic using the rewrite rules that specify the behaviour of the given object-oriented system. In Maude, object states are considered as tuples of the form:

$$\langle Id: C \mid \mathrm{at}_1 : v_1 , ..., at_k : \mathrm{v_k} \rangle$$

where *Id* stands for the object's identity and *C* for its class, while at_1 ,..., $\mathrm{at_k}$ denote attribute identifiers with respective current values v_1 , ..., $\mathrm{v_k}$. Messages can be concurrently sent to / received by such object states, and both object and message instances flow together in the so-called configuration, which introduces the basic concept of concurrent object

systems, as multisets governed by the union operator denoted by '_ _ '. The precise definition of this configuration in Maude itself takes the following form:

```
omod Configuration is
        protecting ID        **** provides OId, CId and AId .
        sorts Configuration Object Msg .
        subsorts OId < Value .
        subsorts Attribute < Attributes .
        subsorts Object Msg < Configuration .
        op _: _: AId Value → Attribute .
        op _, _: Attribute Attributes → Attributes [assoc. comm. Id:nil]
        op ⟨ _: _|_ ⟩ : OId CId Attributes → Object.
        op _ _: Configuration Configuration → Configuration [assoc comm id:nil].
endom
```

In Maude, concurrent object-oriented systems can be defined by means of object-oriented modules—introduced by the keyword (*omod* · · · *endom*)—using a syntax more convenient than that of system modules, because it assumes acquaintance with basic entities, such as objects, messages and configurations, and supports linguistic distinctions appropriate to the object-oriented case. In particular, all object-oriented modules implicitly include the above *Configuration* module and assume its syntax. It should be noted that object modules are internally transformed into system modules for execution.

For example, bank accounts can be represented as UML object and class, as shown in Figure C.4. Each account has balance (*Bal*) as an attribute. Three methods are defined (as messages), namely *credit*, *debit* and *transfer*, respectively to deposit or withdraw the corresponding amount of money in or from the account or to transfer funds between two accounts.

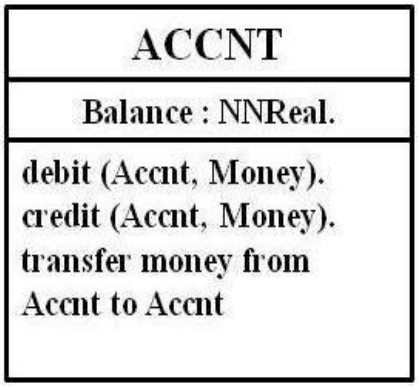


Figure C.4: UML of current account

The following object-oriented Maude module (ACCNT) specifies the concurrent behaviour of such bank accounts.

```
(omod ACCNT is
        protecting REAL.
        class Accnt | bal : NNReal .
        msgs credit debit: OId NNReal → Msg .
        msg transfer_from_to_: NNReal OId OId → Msg .
        vars A B : OId .
        vars M N N': NNReal .
        ********* The Account behaviour.
        rl debit(A,M) <A : Accnt|Bal : N > ⇒  <A : Accnt|Bal : N −M>  if N ≥ M .
        rl credit(A,M) <A : Accnt|Bal : N> ⇒ < A : Accnt|Bal : N +M >.
        rl transfer M from A to B  <A : Accnt|Bal : N >  ⇒ <B : Accnt|Bal : N′>
                <A : Accnt|Bal : N −M> <B : Accnt|Bal : N′ +M>  if N ≥ M .
endom)
```

Classes are defined with the keyword *class*, followed by the name of the class *C* and by a list of attribute declarations separated by commas. Each attribute declaration has the form *a: S*, where *a* is an attribute identifier and *S* is the sort in which the values of the attribute range. That is, class declarations have the class form: <C | a_1 : s_1 , · · · , a_n : s_n > . In this example, the account class (*Accnt*) has only one attribute (*bal)*, which is declared to be a value of type NNReal (non-negative real number), as <A: Accnt |Bal: N>.

The syntax for message declarations is similar to the syntax for the declaration of operators, using keywords *msg* and *msgs*, and having as result sort *Msg* or a subsort of it. In the above example, the three kinds of message—*credit, debit* and *transfer*—are introduced by the keyword *msg* and their resulting sorts are *Msg*.

The effect of messages on targeted object states is modelled using rewrite rules. In the general case, such rules result in (attribute) state changes to some participating objects, the creation / deletion of some objects, the absorption of related messages and the appearance of some new messages.

The debit rule, for instance, says that when an account state receives a debit message, *debit (A,M)*, the next state results in a decrease in the balance by the corresponding amount of money, under the condition that the current balance suffices.

The rewrite rules specify in a declarative way the behaviour associated with the credit, debit and transfer messages. The multiset structure of the configuration provides the top-level distributed structure of the system and allows concurrent application of the rules. To illustrate the above *ACCNT* account module, the following simple account configuration

ACNT-CONF, which consists of three *accounts*, two *debits*, two *credits* and one *transfer*, is proposed. The rewrite rule executions are controlled by the *default strategies*.

(omod ACNT-CONF is
ex ACCNT
ops A_1 A_2 A_3 **→ Oid .**
op AcCfCp → Configuration .
eq AcCfCp = < A_1 **: Accnt|Bal : 500 > debit(**A_1 **, 200) <** A_2 **: Accnt|Bal : 100 > credit(**A_2 **, 50)**
credit(A_1 **, 300)debit(**A_2 **, 300) <** A_3 **: Accnt|Bal : 300 > (transfer200from**A_1 **to**A_3 **) .**
endom)

Figure C.5 shows the effect of the rewriting process on the message and account instances of this configuration as well as the final result (final configuration).

Maude> rew in ACNT-CONF : AcCfCp .
ResultObject :< A_1 **: Accnt|Bal : 400 ><** A_2 **: Accnt|Bal : 150 ><** A_3 **: Accnt|Bal : 500 > debit(**A_2 **,300)**

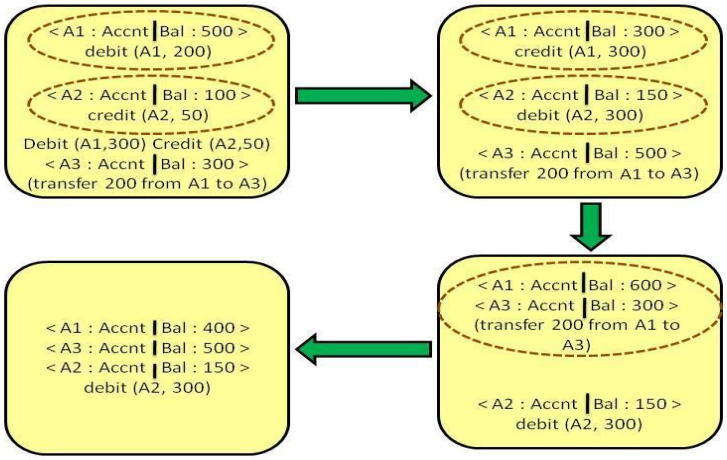


Figure C.5: Concurrent rewriting of bank accounts

C.3.3 Transforming Object-Oriented Modules into System Modules

Although Maude's object-oriented modules provide convenient syntax for programming object-oriented applications, for efficiency purposes their semantics needs to be reduced to that of system modules. In fact, each object-oriented module (*omod* · · · *endom*) can be translated into a corresponding system module *mod* · · · *endm* whose semantics is by definition that of the original object-oriented module.

However, although Maude's object-oriented modules can in this way be reduced to system modules, there are of course important conceptual advantages provided by the syntax of object-oriented modules, because it allows the user to think and express his/her thoughts in

object-oriented terms whenever such a viewpoint seems best suited to the problem at hand. These conceptual advantages would be partially lost if only system modules were provided.

In the translation process, the most basic structure shared by all object-oriented modules is made explicit by the *Configuration* system module defined in the previous subsection. The translation of a given object-oriented module extends this structure with the class, messages and rules introduced by the module. For example, the following system module, *ACCNT♯,* is the translation of the *ACCNT* module introduced earlier.

```
1.mod ACCNT♯ is
2.protecting REAL .
3.including Configuration.
4.sort Accnt .
5.subsort Accnt < Cid .
6.op Accnt : → Accnt .
7.op bal: _: NNReal → Attribute .
8.ops credit debit: OId NNReal → Msg .
9.op transfer_from_to_:NNReal OId OId→Msg.
10.vars A B : OId .
11.Vars M N N' : NNReal .
12.Vars ATTS1 ATTS2 : Attributes .
13.Vars Acnt Acnt1 : Accnt .

********* The Account behavior.

14.rl debit(A,M) <A : Acnt|Bal : N,AT TS1 >
  ⇒ <A : Acnt|Bal : (N −M), AT TS1 > if N ≥ M.

15.rl credit(A,M) <A : Acnt|Bal : N,AT TS1>
    ⇒ <A : Acnt|Bal : (N +M),AT TS> .

16.rl transfer M from A to B <A: Acnt|Bal:N, ATTS1> <B : Acnt1|Bal : N′,AT TS2>
  ⇒ <A : Acnt|Bal : (N −M),AT TS1>
   <B : Acnt1|Bal : (N′+M),AT TS2 > if N ≥ M
.
Endm.
```

The processes of transformation from object-oriented to system module is as follows:

The module *Configuration* is imported.

For each class a declaration of the form *class* C | a_1 : s_1 , · · · , a_n : s_n has to be accompanied by a subsort *C* of sort *Cid,* by a constant *C* of sort *C* and by operations a_i : _ : s_i → *Attribute_sort,* for each attribute a_i .

For each subclass relation *C* < *C′* a subsort declaration *subsort C* < *C′* is introduced.

The rewrite rules are modified to make them applicable to all objects of the given classes and of their subclasses. That is, each object < *O* : *C* | · · · > appearing in a rule is translated into <

O : *X* | · · · , *Atts* >, where the new variable *X* is declared of sort *C* and the new variable *Atts* has sort attributes.

C.4 Reflection and Internal Strategies

Informally, a reflective logic is one of which important aspects of the metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretical aspects. In other words, it is a logic which can be faithfully represented by itself.

As mentioned at the beginning of this chapter, rewriting logic is reflective, in the sense of being able to express its own metalevel at the object level. Reflection is systematically exploited in Maude, by endowing it with powerful metaprogramming capabilities, including both user-definable module operations and declarative strategies to guide the deduction process. Since a naive implementation of reflection can be expensive in both time and memory use, a good implementation must provide efficient ways of performing reflective computations. This section explains how this is achieved in Maude through the predefined *META-LEVEL* module.

C.4.1 Reflection in Maude

Rewriting logic is reflective [66] in essence. That is, any rewrite theory can be (meta-) represented at a higher level and be reasoned on like data manipulation. This reflective property has been nicely defined using the following abstraction, where universal $\mathcal{U}$ represents the (meta)theory at the higher level in which any rewrite system and / or terms can be manipulated as (meta)data.

$$\mathcal{R} \Rightarrow t \rightarrow t' \Leftrightarrow \mathcal{U} \Rightarrow \langle \overline{R}, \overline{t} \rangle \rightarrow \langle \overline{R}, \overline{t'} \rangle$$

META-LEVEL has sorts *Term* and *Module*, so that the representations of a term *t* and of a module $\mathcal{R}$ are, respectively, a term $\bar{t}$ of sort *Term* and a term $\bar{R}$ of sort *Module*.

The *META-LEVEL* module also provides key functions for rewriting and evaluating terms at the metalevel, namely, upModule, upTerm, downTerm, metaReduce, metaRewrite,

metaApply, metaXapply, etc. Each of these reflection operation primitives is detailed as follows.

Moving between reflection levels: upModule, upTerm and downTerm. The operation upModule takes as arguments the meta-representation of the name of $\mathcal{R}$ and a Boolean value *b*, and returns the respective meta-representation of the module $\mathcal{R}$.

— If the second argument of these functions is true, then the resulting meta-representation will include the corresponding statements that $\mathcal{R}$ imports from its submodules;

— If the second argument of these functions is false, the resulting meta-representation will contain only the meta-representation of the statements declared in $\mathcal{R}$.

op upModule : Qid Bool ~ > Module [special (...)] .

The polymorphic functions upTerm and downTerm can move terms between the reflection levels. upTerm transfers a term into its meta-representation. Therefore, the upTerm function can be used to avoid the cumbersome task of writing the meta-representation of a term or of a module. It takes a term *t* and returns the meta-representation of its canonical form. The result of a metalevel computation that may use several levels of reflection can be a term or module meta-represented one or more times, which may be hard to read. The downTerm function can be used to display the output in a more readable form, which in sense inverses to upTerm, since it returns the term from its meta-representation. The downTerm function takes the meta-representation of a term *t* as its first argument and a term *t´* as its second argument, returning the canonical form of *t*, if *t* is a term in the same kind as *t´*; otherwise, it returns the canonical form of *t´*.

op upTerm : Universal → Term [poly special (...)] .

op downTerm : Term Universal → Universal [poly special (...)] .

Simplifying meta-terms: metaReduce. The function metaReduce takes as arguments the meta-representation of a module $\mathcal{R}$ and the meta-representation of a term *t* in that module, returning the meta-representation of the fully reduced form of the term *t* using the equations in $\mathcal{R}$, together with its corresponding sort or kind:

op metaReduce : Module Term → ResultPair [special (...)] .

op { _ , _ } : Term Type → ResultPair [ctor] .

The reduction strategy used by metaReduce coincides with that of the reduce command mentioned above.

Rewriting at the reflection-level: metaRewrite. The (partial) operation metaRewrite is entirely analogous to metaReduce, except that instead of using only the equational part of a module, it now uses both the equations and the rules to rewrite the term. The function metaRewrite takes as arguments the meta-representation of a module $\mathcal{R}$, the meta-representation of a term *t*, and a value *b* of the sort Bound, i.e. either a natural number or the constant unbounded. Its result is the meta-representation of the term obtained from *t* after at most *b* applications of the rules in $\mathcal{R}$ using the rewrite strategy, together with the meta-representation of its corresponding sort or kind:

op metaRewrite: Module Term Bound → ResultPair [special (...)].

op { _ , _ } : Term Type → ResultPair [ctor] .

Applying reflection rules: metaApply and metaXapply. The operation metaApply basically takes a term and a rewrite rule in a given module, then rewrites the term once by applying the specified rule. It has this syntax:

op metaApply : Module Term Qid Substitution Nat → ResultTriple?

[special (...)] .

op { _ , _ , _ } : Term Type Substitution → ResultTriple [ctor] .

The first argument, *Module*, is the meta-representation of the module that defines the terms and rules in question. *Term* is the given term to be rewritten, which must match exactly the left-hand side of the rewrite rule applied (modulo associativity, commutativity, etc.) and *Qid* is the name of the rewrite rule to be applied. A set of assignments (possible empty) defines a partial substitution for the variables in those rules. The last argument is a natural number *n* used to enumerate (starting from 0) all the possible solutions of the intended rule application. It returns a tuple of sort *ResultTriple* consisting of a term with the corresponding sort or kind and a substitution.

The operation metaXapply applies a rule to a term in any possible position. The first four arguments are the meta-representation of a module $\mathcal{R}$, the meta-representation of a term *t* in $\mathcal{R}$, a label *l* of some rules in $\mathcal{R}$ and a set of assignments (possibly empty) defining a partial substitution σ for the variables in those rules. The last natural number enumerates the solutions, since there can be different such rewrites with different substitutions and at different positions. The other two numeric arguments indicate the minimum and maximum depth in the term where the application of the rule can take place.

op metaXapply : Module Term Qid Substitution Nat Bound Nat → Result4Tuple? [special (...)] .

op { _ ,_ ,_ ,_ } : Term Type Substitution Context → Result4Tuple

[ctor] .

metaXapply returns a tuple of sort *Result4Tuple* consisting of a term with the corresponding sort or kind, a substitution and the context inside the given term where the rewriting has taken place.

C.4.2 Internal Strategies

As mentioned already, system modules in Maude are rewrite theories that do not need to be Church-Rosser and terminating. Therefore, there is a need to control the rewriting inference process by means of adequate strategies. Using reflection, this can be done with great flexibility by means of internal strategies which can be defined using statements in a normal module in Maude and which can be reasoned about as with statements in any other module.

In fact, there is great freedom to define many different types of strategy, or even many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a particular fixed and closed strategy language. In general, strategies for controlling the application of the rules are defined in extensions of the *META-LEVEL* module using metaReduce, metaApply, metaXapply etc. as building blocks, which are then combined to obtain more complex strategies.

Example: To illustrate the application of strategies, consider again the Account specification and its rules. A logical strategy for executing these rules should consist in the following. First, all debits are to be performed, then all credits are to be performed and finally all

transfers have to be executed. The following module written in reflective Maude, referred to below as *ACCNT-STR*, reflects this strategy at the metalevel.

```
mod ACCNT-STR is
  inc ACNT-CONF .
  protecting META-LEVEL .
  vars debit? credit? transfer? : [Result4Tuple] .
  var T : Term .
  op Compute : Term → Term .
  ceq Compute(T)
     = (if(debit? :: Result4Tuple)
          then getTerm(debit?)
          else if(transfer? :: Result4Tuple)
          then getTerm(transfer?)
          else if(credit? :: Result4Tuple)
          then getTerm(credit?)
       else T fi fi fi)
          if debit? = metaXapply(upModule('ACCNT, false), T,
                 'debit, none, 0, unbounded, 0)
        ∧ credit? = metaXapply(upModule('ACCNT, false), T,
                 'credit, none, 0, unbounded, 0)
        ∧ transfer? = metaXapply(upModule('ACCNT, false), T,
                 'transfer,none, 0, unbounded, 0)
Endm
```

Using the same concrete configuration as in section C.3.2, the rewriting process on such instances in *META-LEVEL* is shown in Figure C.6. Note that to reduce such a configuration at the metalevel, we have first to bring it to the metalevel using the meta-command upTerm. The result is then collected by bringing the result to the usual object level using the meta-command downTerm.

Maude> reduce in ACNT-STR : downTerm(Compute(upTerm(AcCfCp)), 'error) .
ResultObject :< A_1 : Accnt\|Bal : 400 >< A_2 : Accnt\|Bal : 150 > < A_3 : Accnt\|Bal : 500 > debit(A_2 , 300) .

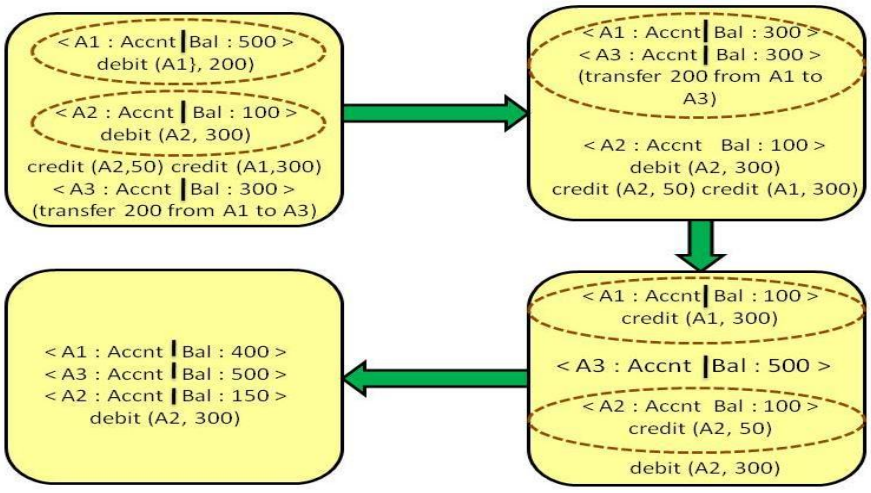


Figure C.6: Strategies to control rule execution

C.5 Overview of Maude's Workstation Environment

Maude Workstation is a programming environment for Maude, written in Java, which makes it executable on different platforms.

In the work reported in this thesis, all Maude extensions, experiences and case-studies have been implemented in this Maude-Workstation environment, a general view of which is given by the screen-shot depicted in Figure C.7. The environment has two main parts: the edition facilities and the Maude emulation area, located respectively in the upper and lower areas.

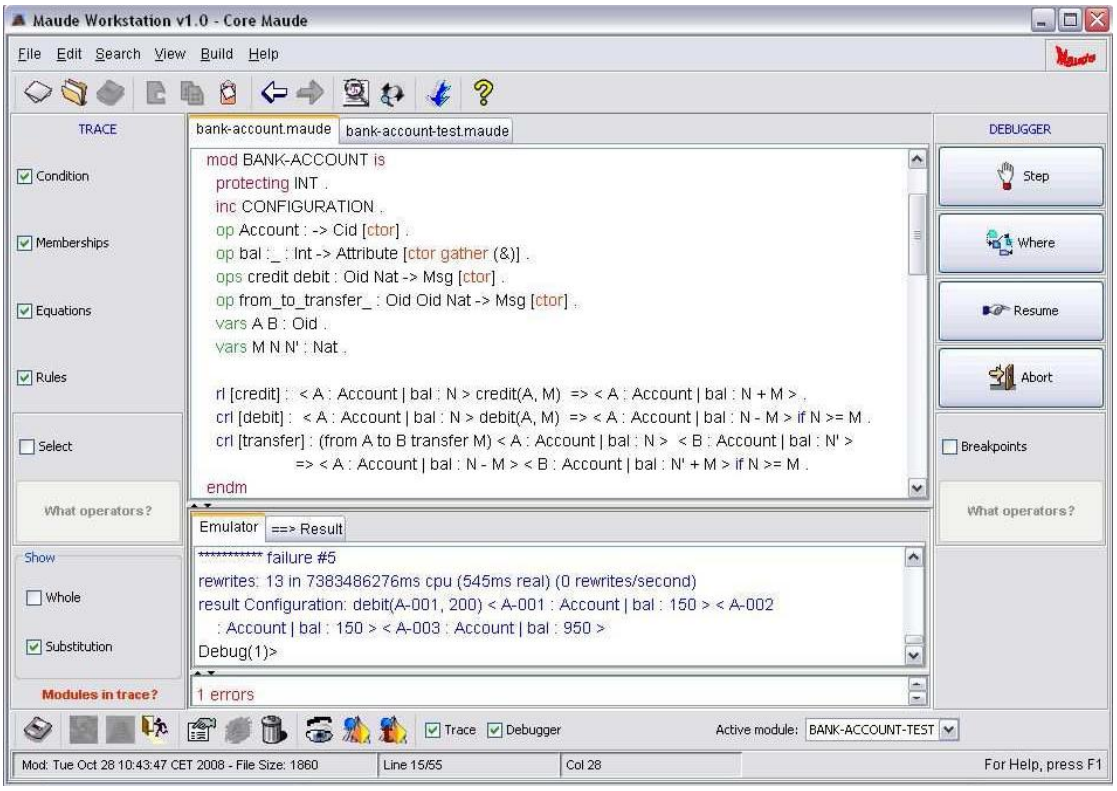


Figure C.7: General view of Maude Workstation

In the edition area, opened files are accessible through a split pane system. The emulator area is composed of three split panels: Emulator, Result and a text frame for syntax mistakes. In the emulator split, the interaction with the Maude interpreter is simulated, allowing the user to work as if he/she were at a traditional Maude console. In the Result split, one can find answers after sending to Maude the commands through the emulator or files from the edition bar. For instance, Figure C.8 shows the rewrite result of the bank account test that is given in the edition area. The frame dedicated to showing syntactic mistakes is located at the bottom

of the environment. It shows any errors found after sending a file to Maude through the editor.

There are also two tool bars (the trace one on the left and the debugger one on the right) which appear whenever the user needs them. They are located vertically on either side of the environment, allowing easy manipulation of the tracing and depuration facilities.

During the process of sending active file to Maude, the text is analysed by Maude and errors are returned through the emulator and the syntactic mistakes frames. The main function of the mistakes frame is to show the Maude error messages in greater detail, indicating the location and type of any mistakes**.**

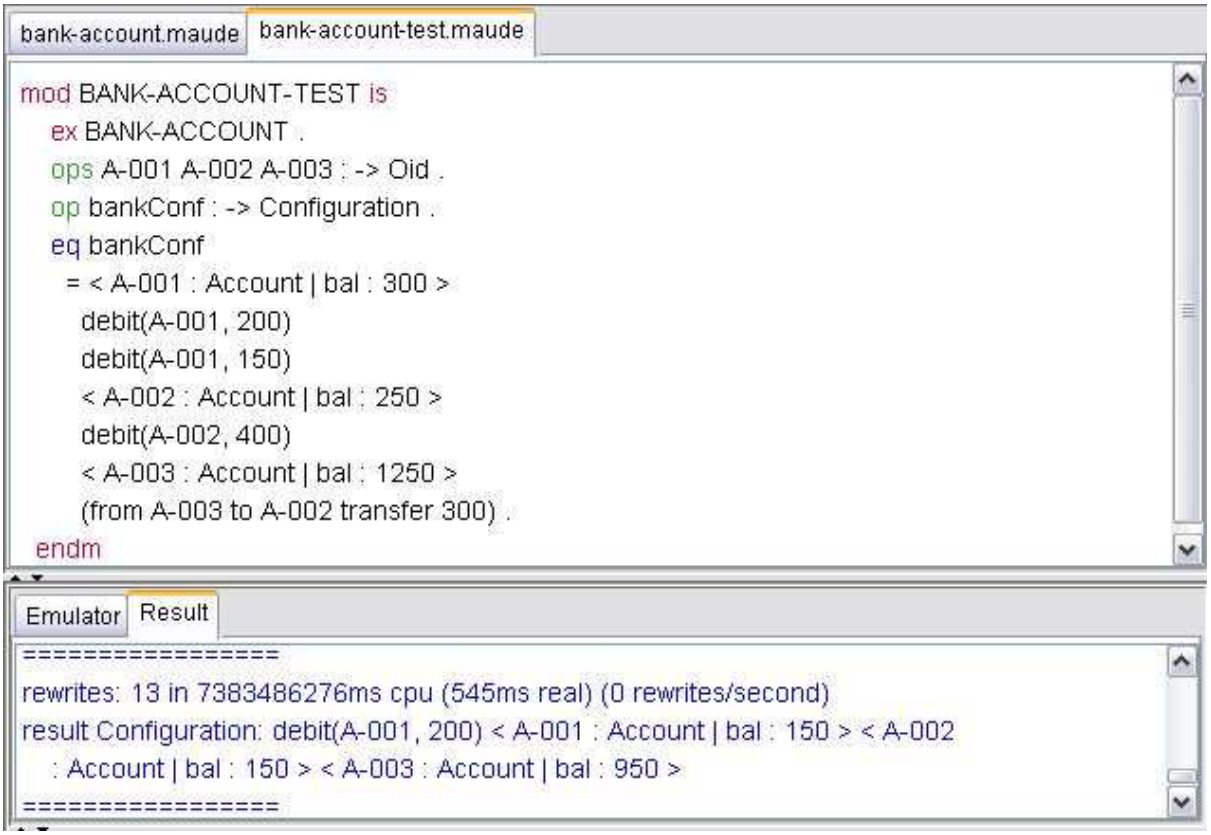


Figure C.8: The result split panel of Maude Workstation

Maude Workstation has an indicator in the top right corner whose function is to show at all times whether Maude is running in the environment. It changes colour from black to red when the Maude process starts and keeps moving while communication is established. It also indicates whether Core Maude or Full Maude is running.

Maude Workstation also has a sequence of toolbars giving full information on the specifications of Core and Full Maude; this is the Show Information window, depicted in Figure C.9. It allows information from the modules stored in the local database to be recovered without having to communicate with the Maude process. Users select the module from the menu in the top right corner to request information such as the operators of a specific module, its membership, equations, classes, messages, kinds, etc., plus the module itself if the user needs it.

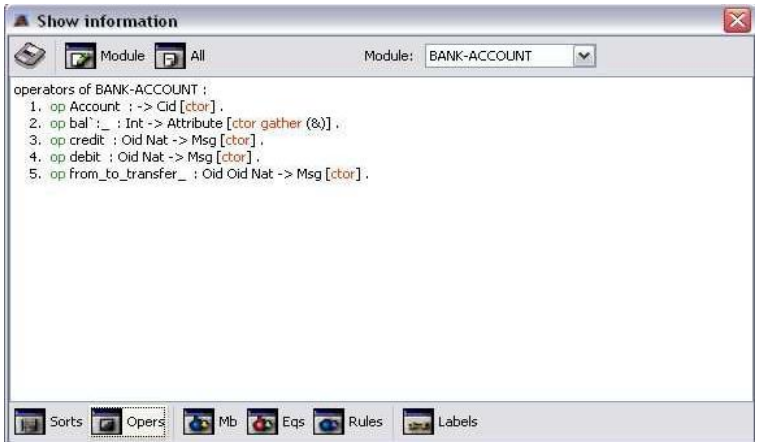


Figure C.9: The Show Information window

It is also possible to send reduction and rewriting commands to Maude through the dialogue window shown in Figure C.10, which saves the history of commands.

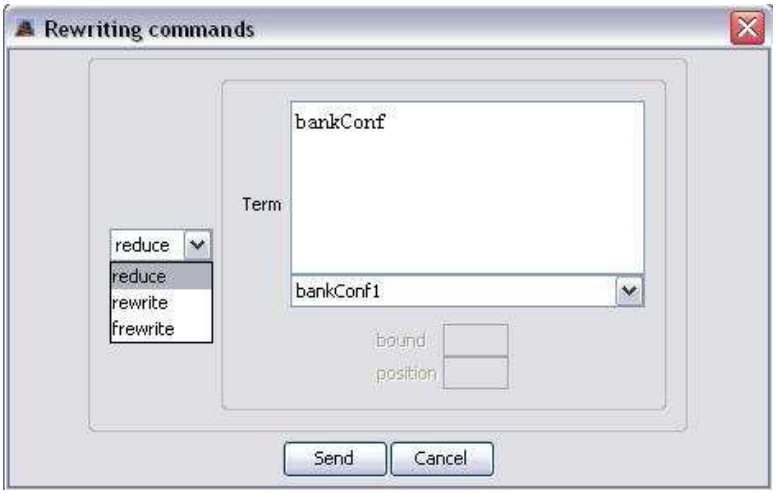


Figure C.10: Window for sending reduction or rewriting commands to Maude